SESSION 2

iOS DEVELOPMENT

REMINDERS

- Please join these Slack channels:
  - #section-2 for announcements and questions specific to our section
  - #office-hours for announcements and questions specific to office hours
  - #assignment-2 for HW 2 questions and clarifications

WELCOME TO THE WORLD OF

UIKit

WHAT IS UIKit?

- UI stands for User Interface
- Your app's user interface includes:
  - Labels, buttons, switches, text fields
  - Navigation between screens
  - Gestures like tapping and swiping

WHAT IS UIKit?

- Things that are not part of the user interface:
  - Downloading and uploading data
  - Database operations
  - Analytics
- These things may or may not have an impact on the user interface, but are not considered part of it.

WHAT IS UIKit?

- In iOS development, Kit is often used to indicate that something is a framework
  - E.g., UIKit, MapKit, HomeKit, ARKit

WHAT IS A FRAMEWORK?

- Code written by somebody else
- An abstraction around common concerns
  - This prevents developers from having to re-invent the wheel
- Code in a framework can be extended but not modified

CONSTRUCT AND MANAGE A GRAPHICAL, EVENT-DRIVEN USER INTERFACE

UIKit Documentation

WORKING WITH

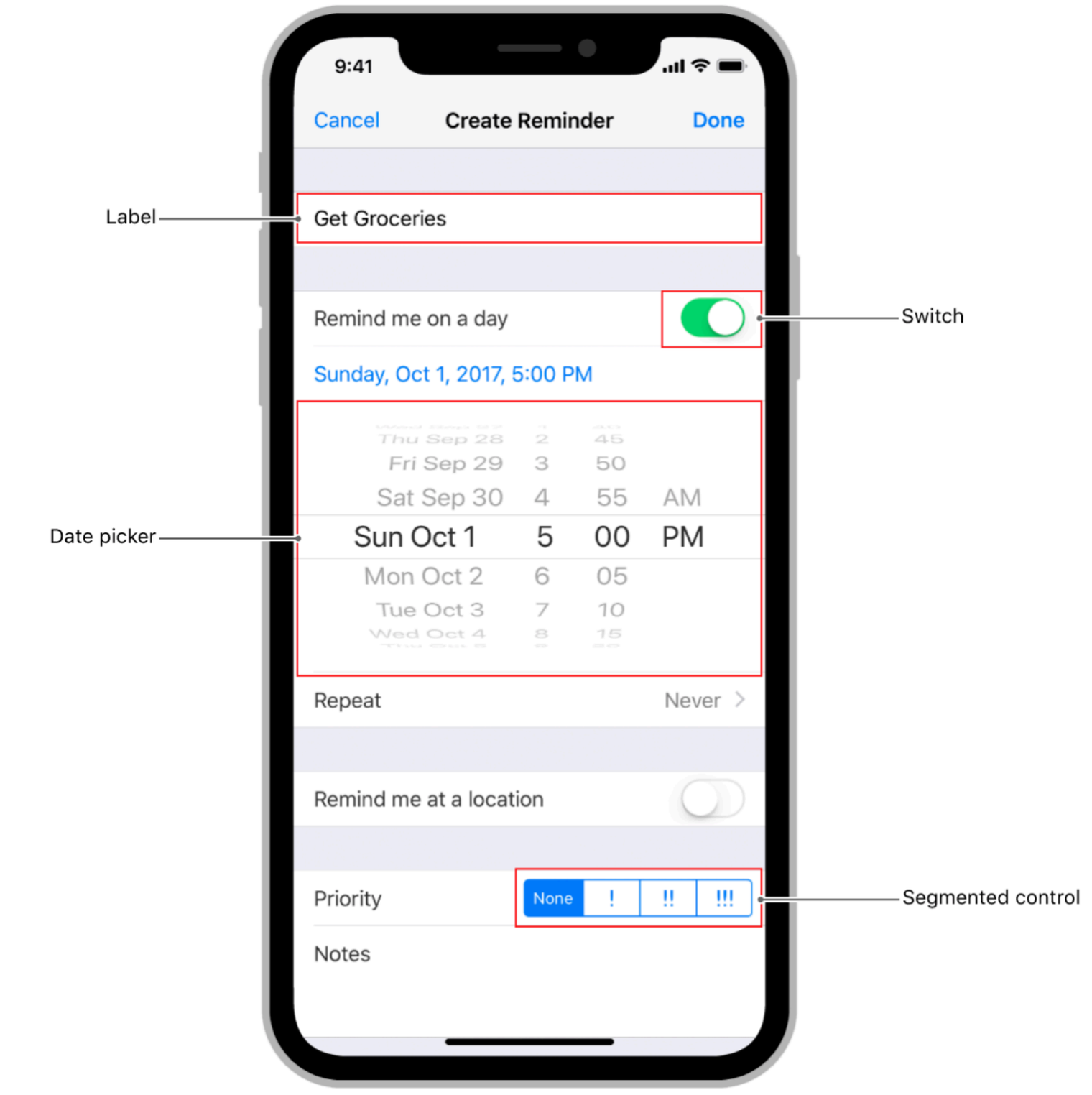
VIEWS & CONTROLS

VIEWS AND CONTROLS ARE THE VISUAL BUILDING BLOCKS OF YOUR APP'S USER INTERFACE

UIKit Documentation

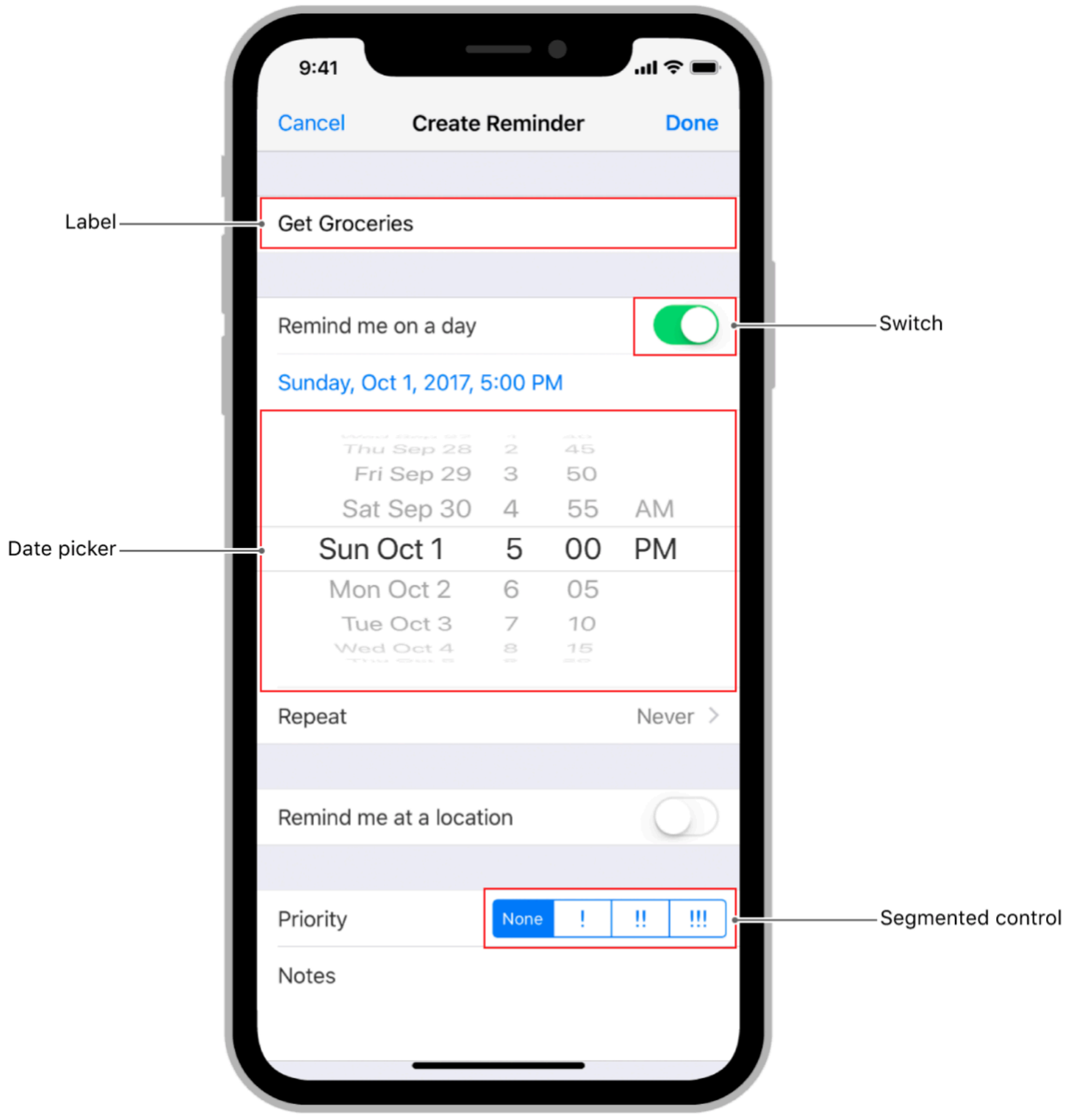
EXAMPLES

- UIKit provides many common views and controls



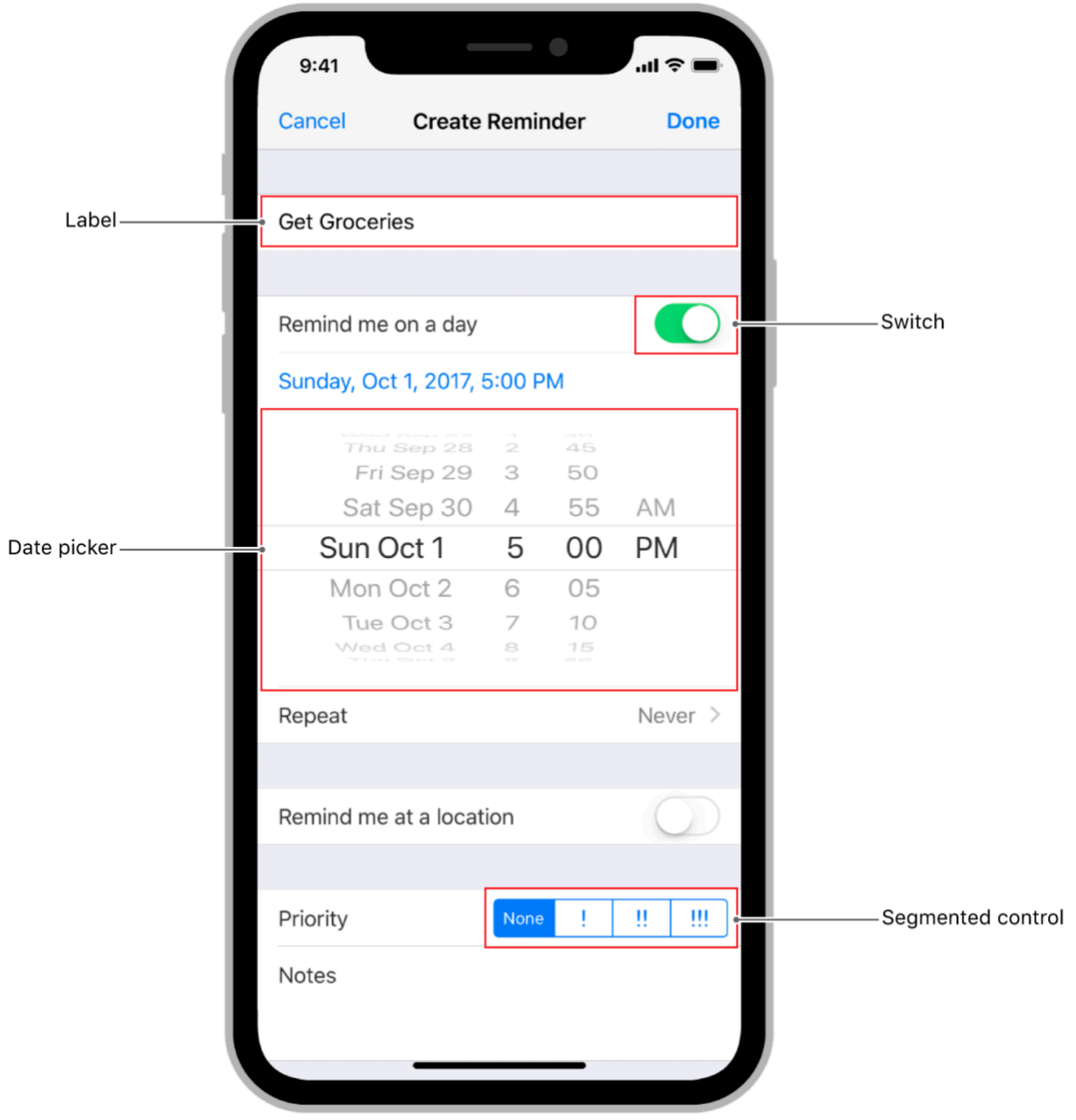
VIEW VS. CONTROL

- View: any subclass of `UIView`
- May or may not be interactive
- Example: label, image view



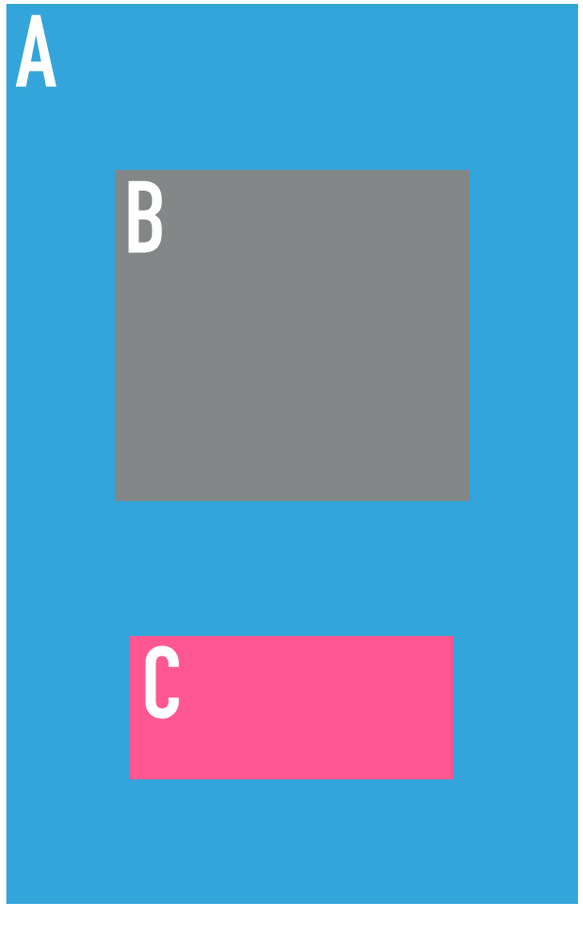
VIEW VS. CONTROL

- Control: a special kind of view
- Enables some type of user interaction
- Example: switch, slider, segmented control



SUPERVIEW & SUBVIEW

- A view can contain one or more subviews
- A view may have one or zero superviews



- A is a superview of B and C
- B and C are subviews of A
- B and C are siblings to each other

WHAT CAN A VIEW DO?

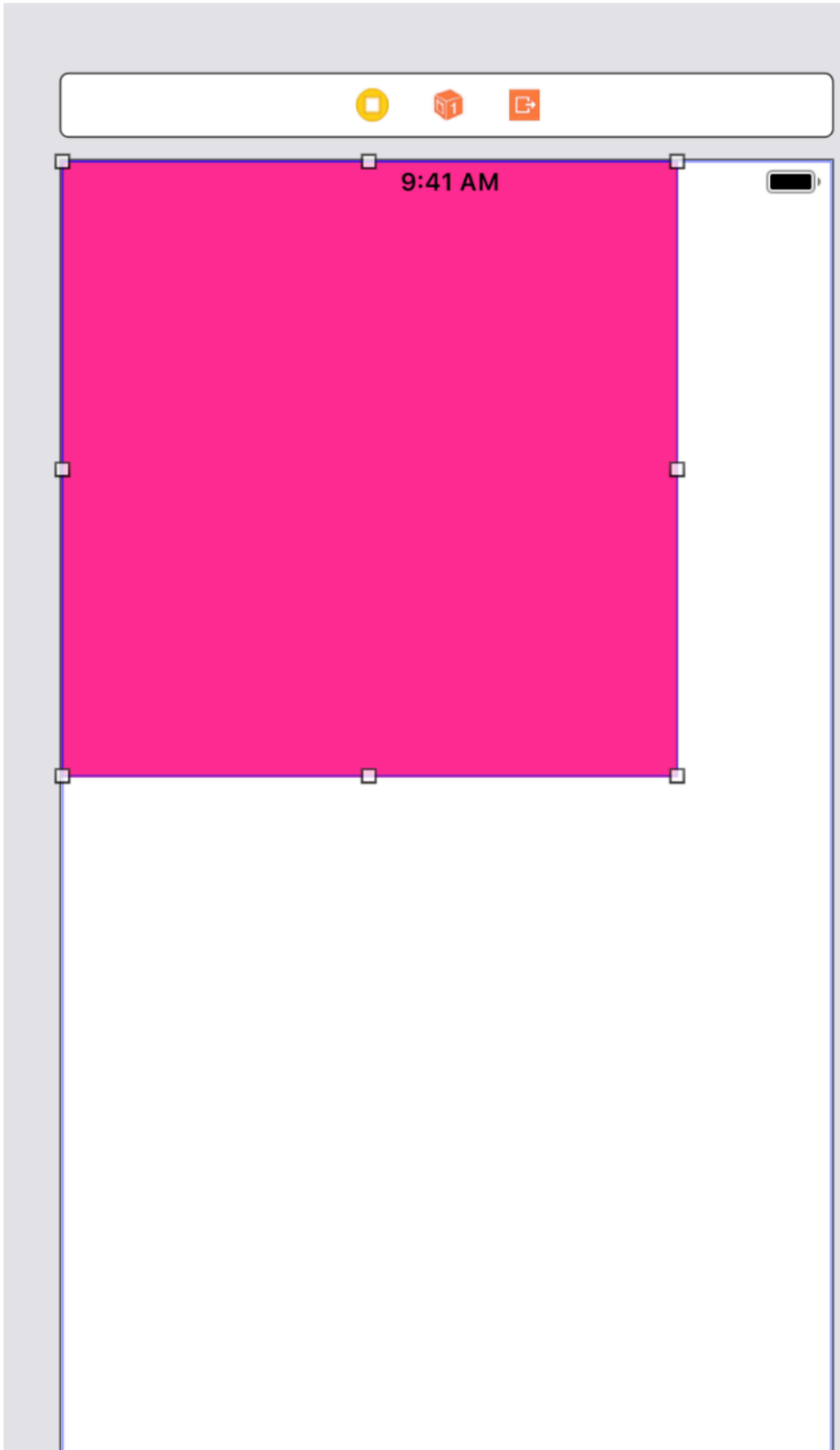
- Drawing and animation
- Layout and subview management
- Event handling (i.e., responding to user interactions such as tapping, swiping, sliding, editing, etc.)

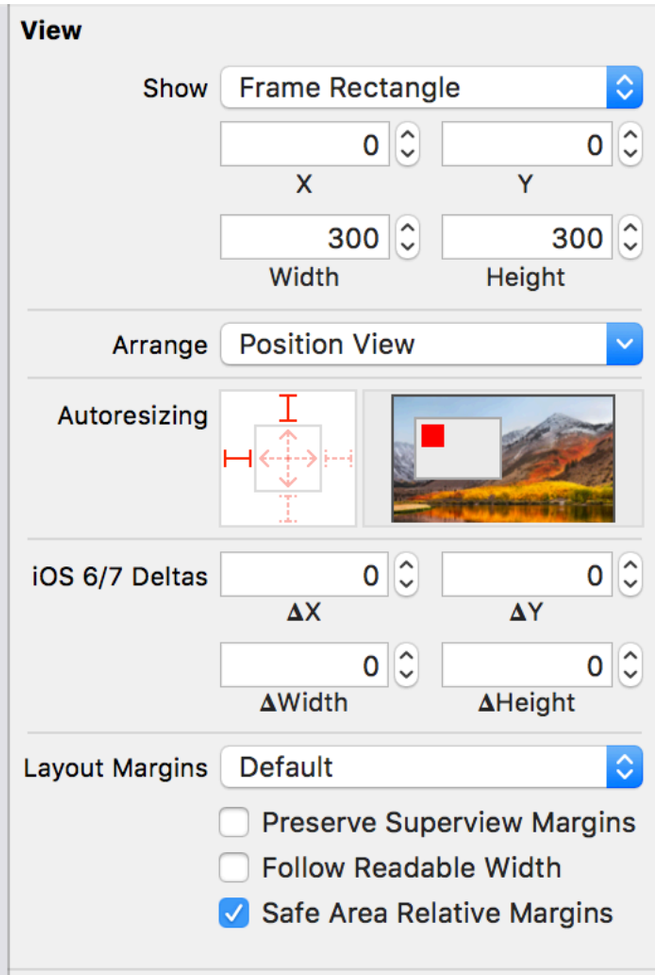
VIEW GEOMETRY

- Every view has a `frame`, made up of:
  - x-coordinate
  - y-coordinate
  - width
  - height
- These are measured in points, not pixels

VIEW GEOMETRY

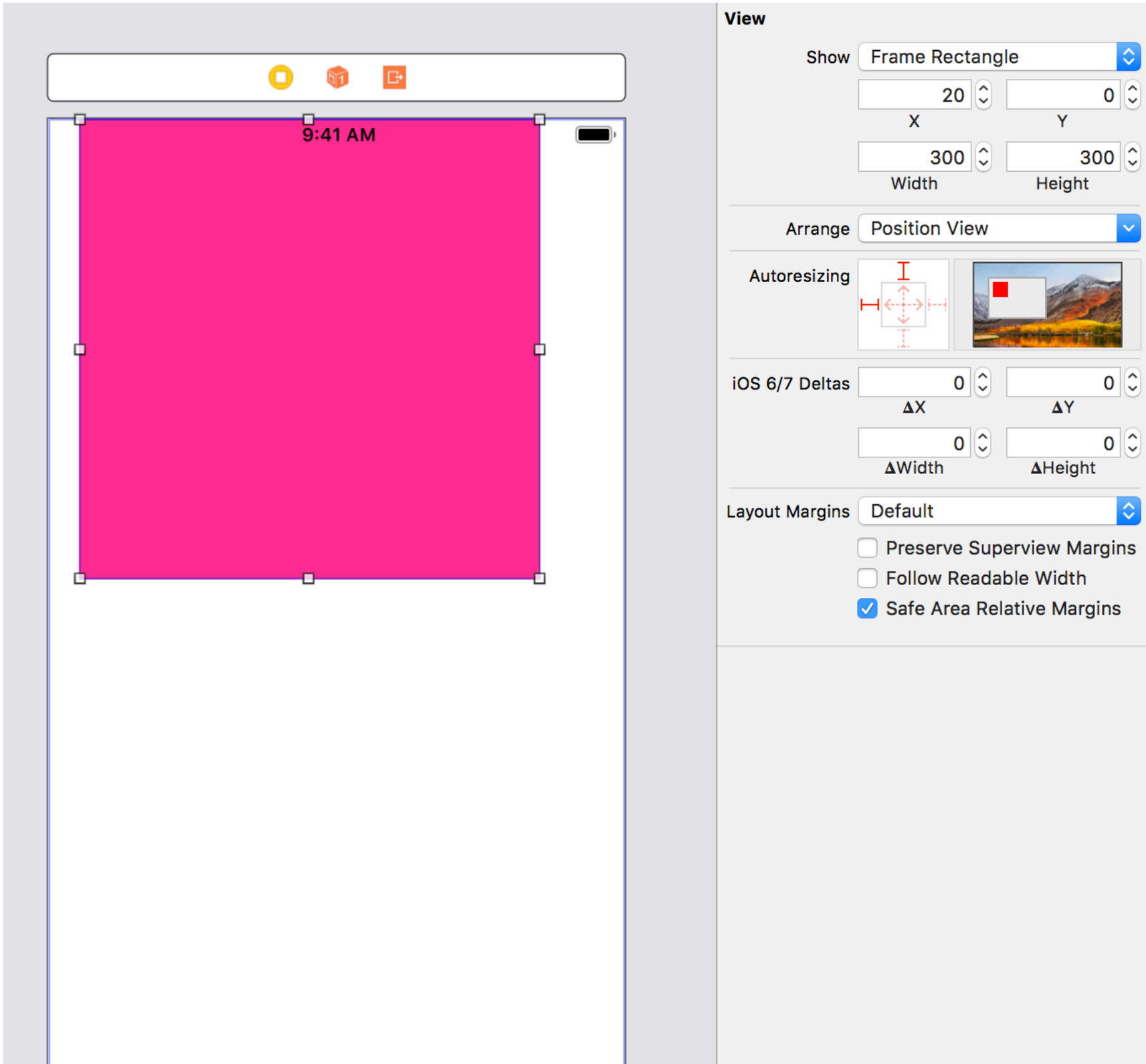
- (0, 0) is in the top left corner of the screen





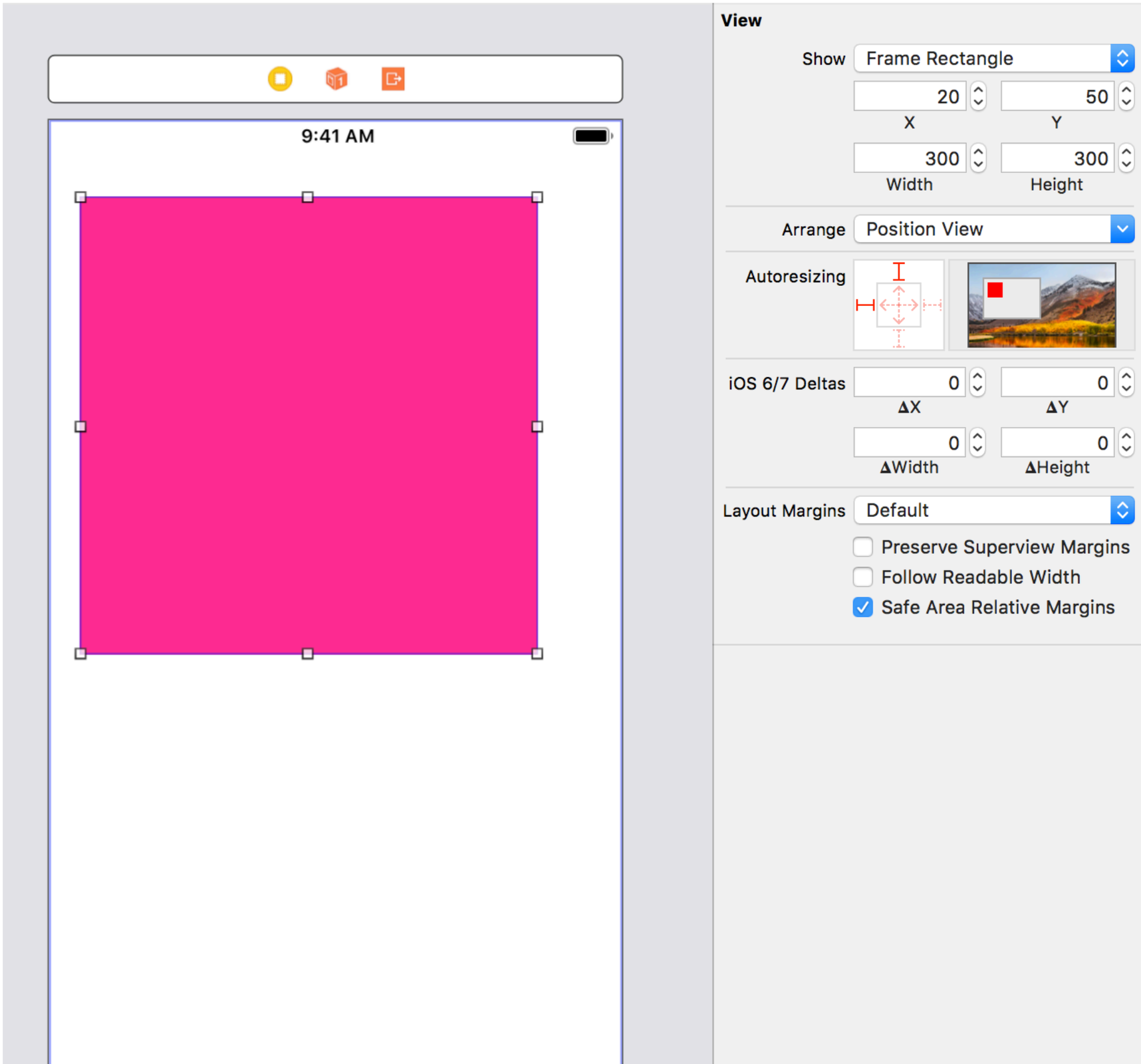
VIEW GEOMETRY

- Increasing `x` by 20 moves the view to the right



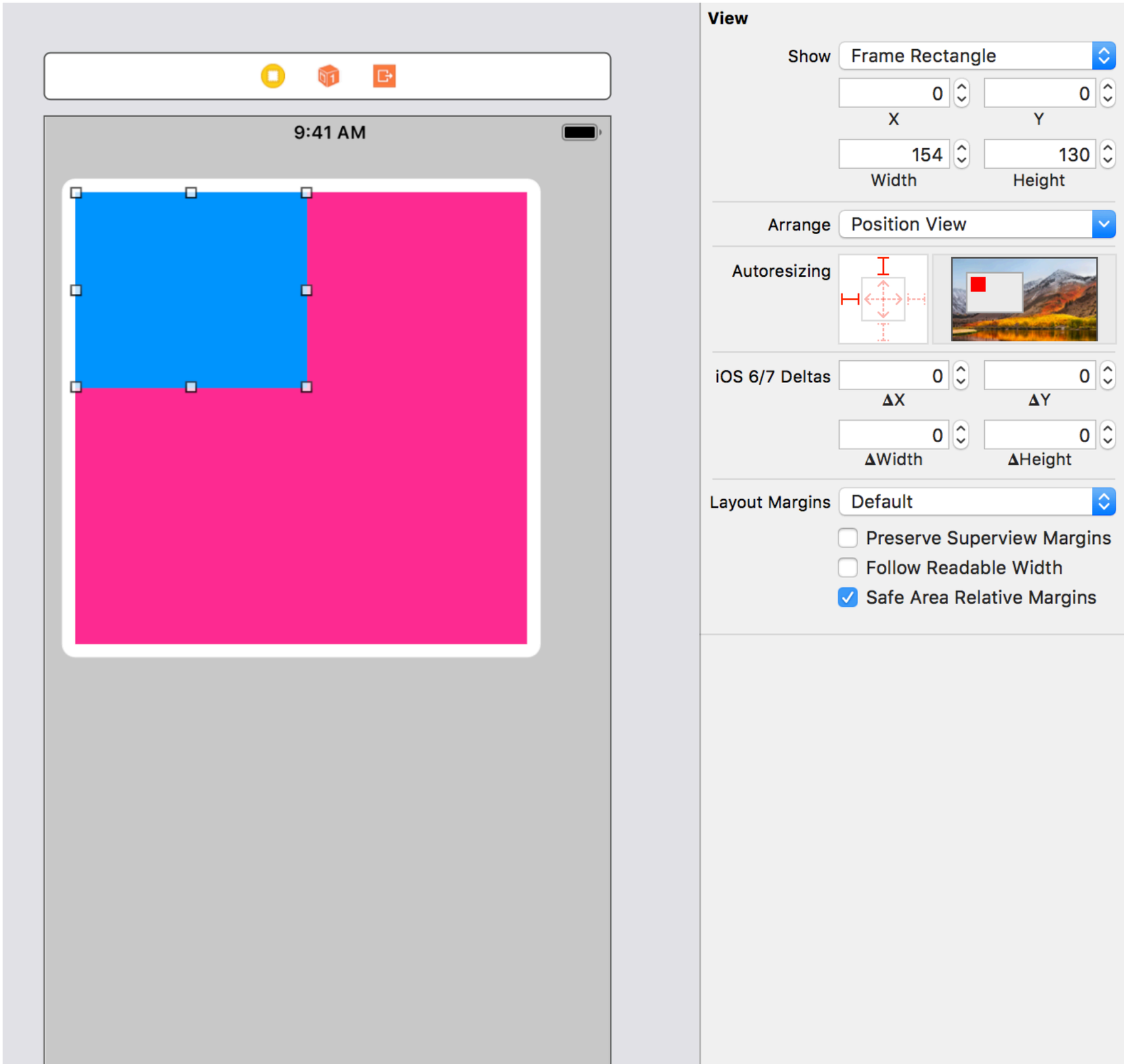
VIEW GEOMETRY

- Increasing y by 50 moves the view down



VIEW GEOMETRY

- The `frame` is always relative to the superview
- The blue view's origin is (0, 0) not (20, 50)



VIEW GEOMETRY

- Every view also has a `bounds` property, made up of:
  - x-coordinate
  - y-coordinate
  - width
  - height

FRAME VS BOUNDS

- `frame`: relative to the view's superview
- `bounds`: relative to the view itself
- Example:
  - frame = (x: 20, y: 50, width: 300, height: 300)
  - bounds = (x: 0, y: 0, width: 300, height: 300)
- `bounds` is typically used for custom drawing

PROPERTIES OF EVERY VIEW

- `backgroundColor`: the color of the view
- `isHidden`: whether the view is visible on screen
- `alpha`: the transparency of the view
- `isUserInteractionEnabled`: whether the user can interact with the view
- `tag`: an integer that you can use to identify the view

PROPERTIES OF EVERY VIEW

- `frame`: the view's dimensions, from the superview's perspective
- `bounds`: the view's dimensions, from it's own perspective
- `center`: a point at the center of the view

PROPERTIES OF EVERY VIEW

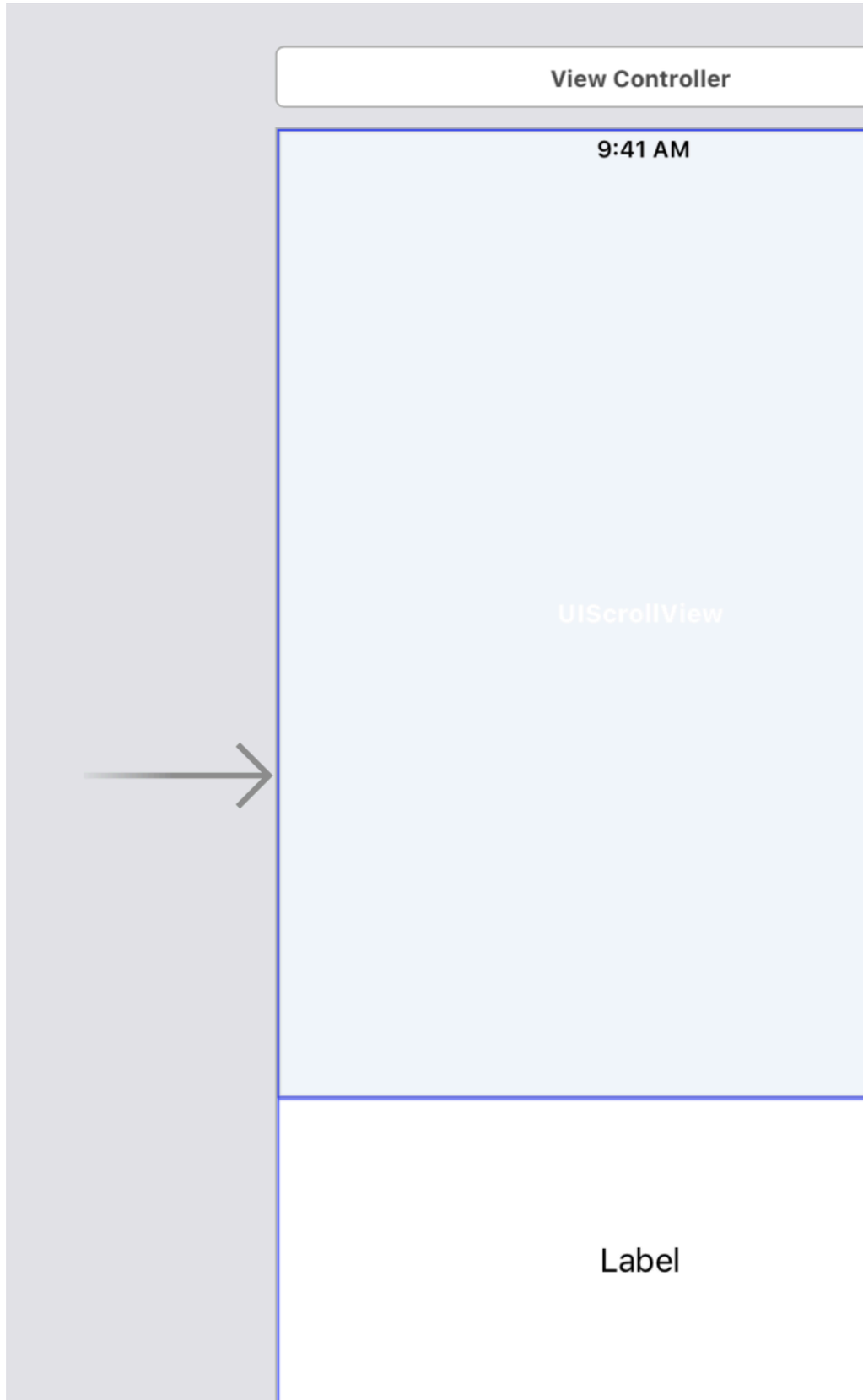
- `superview`: the view's parent
- `subviews`: the view's children
- Methods to modify layout of subviews:
  - `addSubview(_:)`
  - `insertSubview(_:at:)`
  - `bringSubviewToFront(_:)`
  - `sendSubviewToBack(_:)`

INTRODUCTION TO

VIEW CONTROLLERS

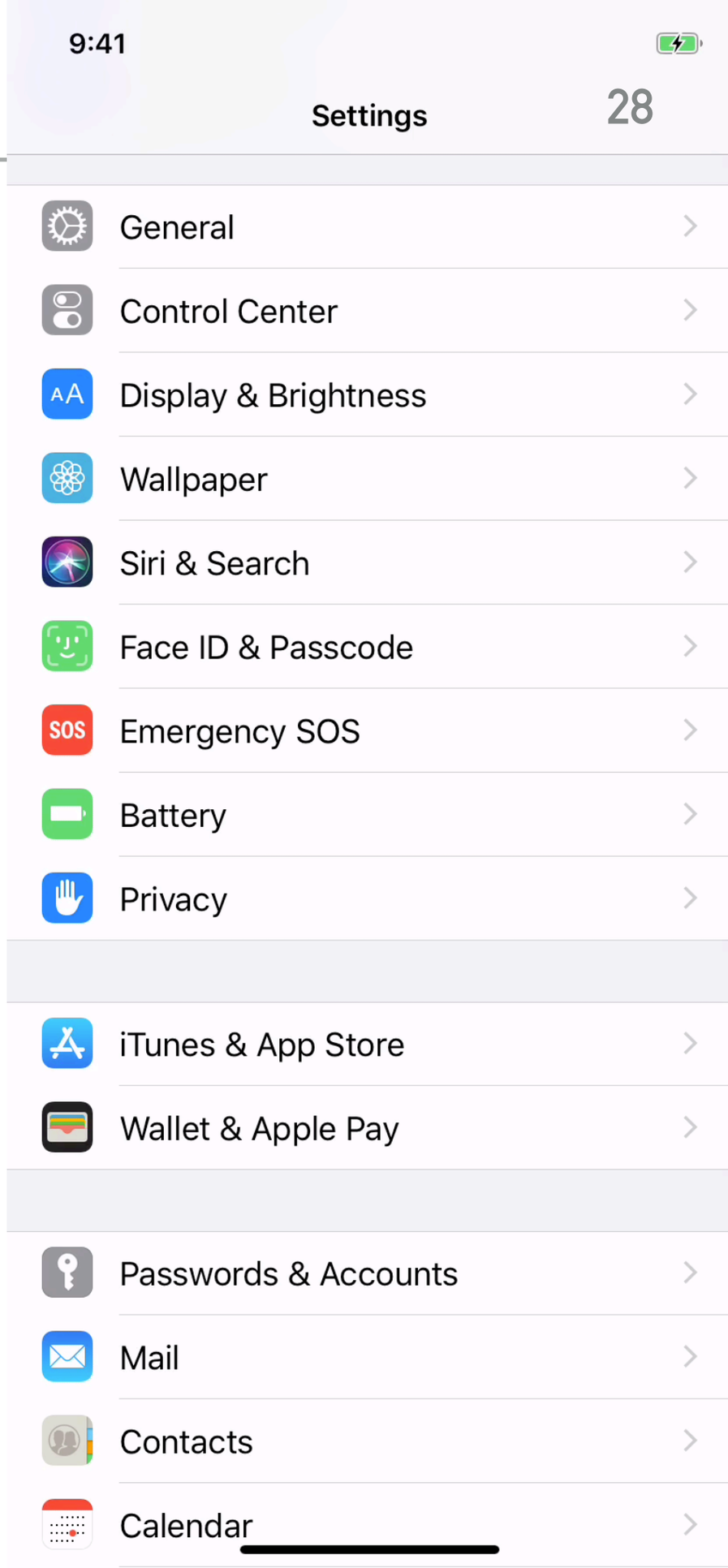
VIEW CONTROLLERS

- Every app has one or more view controllers
- Arrow in storyboard indicates which one will be displayed first



VIEW CONTROLLERS

- Usually, each new “screen” is a view controller
- Every view controller is a subclass of `UIViewController`



LIFECYCLE OF A VIEW CONTROLLER

- Lifecycle = initialization to destruction
- `UIViewController` has several methods that are called throughout its lifecycle
- Your subclass can `override` these methods and implement custom behavior

LIFECYCLE METHODS

- `func viewDidLoad()`
- `func viewWillAppear(_ animated: Bool)`
- `func viewDidAppear(_ animated: Bool)`
- `func viewWillDisappear(_ animated: Bool)`
- `func viewDidDisappear(_ animated: Bool)`

LIFECYCLE METHODS

- `viewDidLoad()`
  - Called when the content view is created and loaded from storyboard
  - IBOutlets guaranteed to have valid values
  - Called only once

LIFECYCLE METHODS

- `viewWillAppear(_ animated: Bool)`
  - Called just before the content view is added to the app's view hierarchy
  - May be called multiple times as the view controller appears and disappears

LIFECYCLE METHODS

- `viewDidAppear(_ animated: Bool)`
  - Called just after the content view is added to the app's view hierarchy
  - May be called multiple times as the view controller appears and disappears

LIFECYCLE METHODS

- `viewWillDisappear(_ animated: Bool)`
  - Called just before the content view is removed from the app's view hierarchy
  - May be called multiple times as the view controller appears and disappears

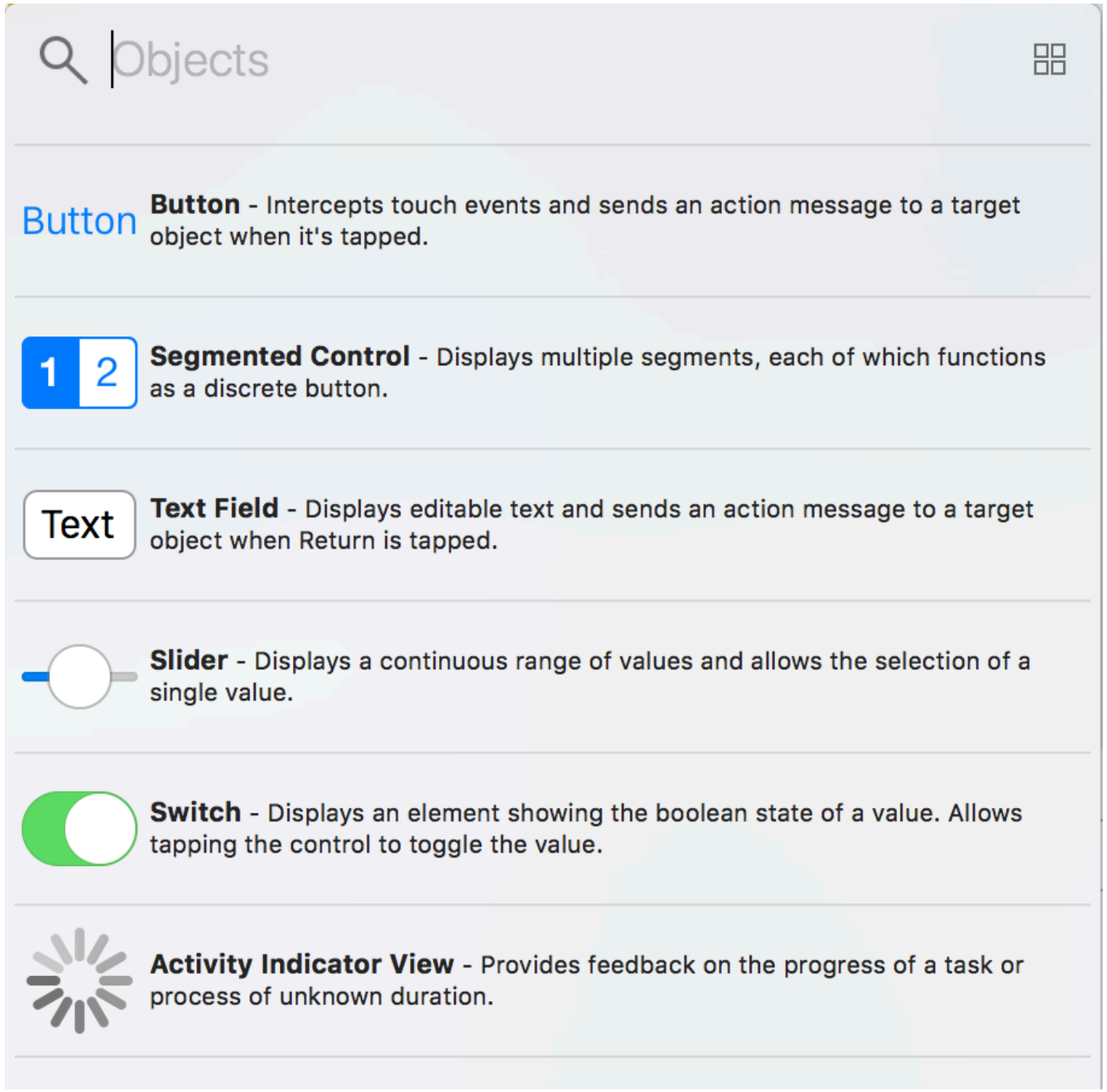
LIFECYCLE METHODS

- `viewDidDisappear(_ animated: Bool)`
  - Called just after the content view is removed from the app's view hierarchy
  - May be called multiple times as the view controller appears and disappears

USING UIKit

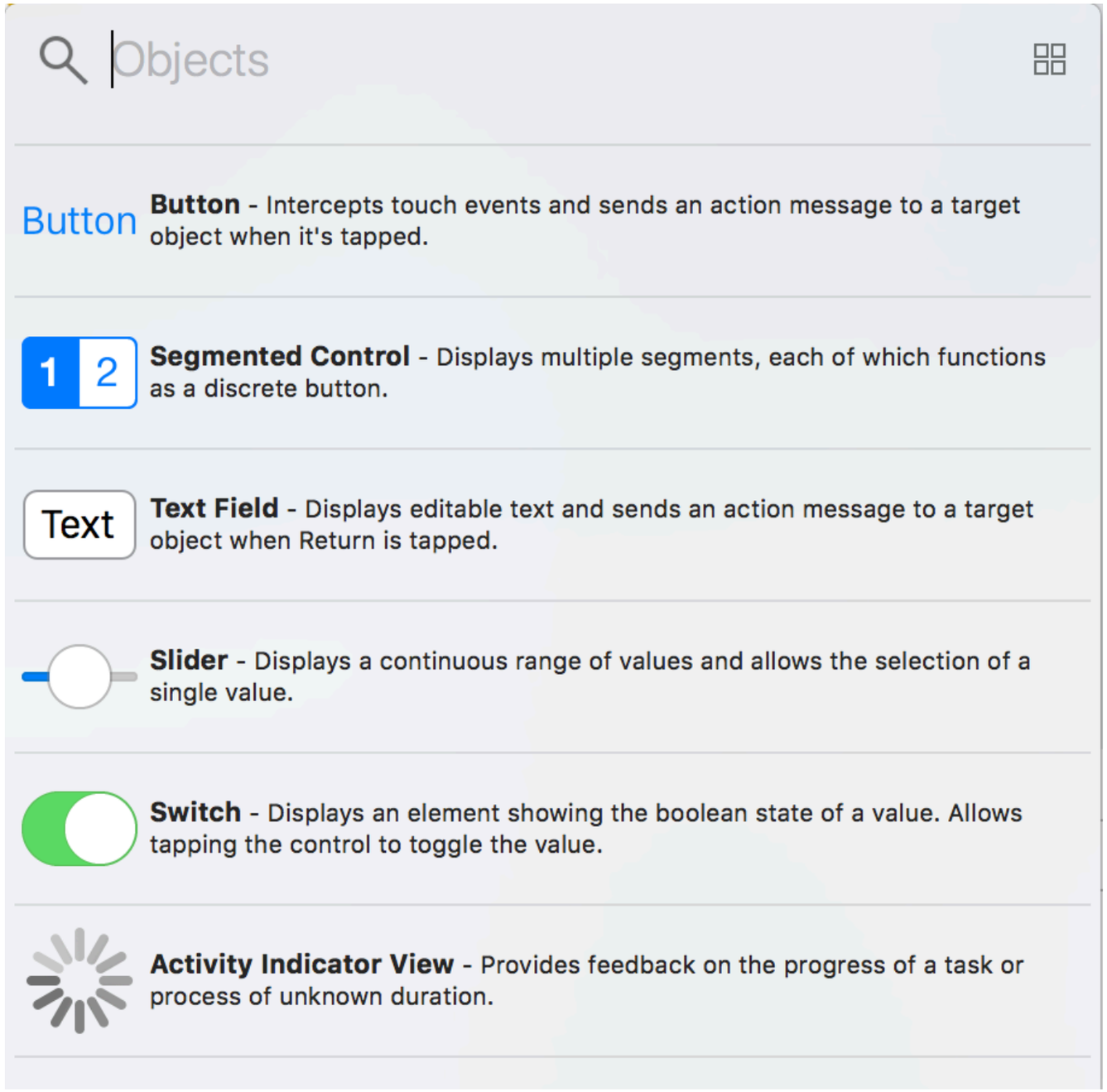
HOW DOES IT ALL COME TOGETHER?

- Apple has created lots of views and controls
- How do you connect them to YOUR code?



HOW DOES IT ALL COME TOGETHER?

- There are many different patterns
- We'll discuss three



PATTERN #1

CLOSURES

WHAT IS A CLOSURE?

- A block of code that accepts input parameters and may return a value
- What does this sound like?

OTHER NAMES FOR CLOSURES

- Closures are also known as:
  - Blocks
  - Lambdas
  - Anonymous functions
  - Callbacks
  - Completion handlers

A CLOSURE IS A FUNCTION

- Key point: A closure is a function.
- Keep this in mind as we discuss the many different ways to write a closure.

SORTED(BY:) METHOD

- Closures are a feature of Swift.
- We'll explore closure syntax using Swift's `sorted(by:)` method, which sorts the elements of an array according to a predicate.
- Then, we'll discuss how to use closures with UIKit.

SORTED(BY:) METHOD

```
let cities = ["Albuquerque", "Chicago", "Boston"]

func reverseAlphabetical(_ s1: String, _ s2: String) -> Bool {
    return s1 > s2 // returns true if s1 comes after s2 in the alphabet
}

let reversed = cities.sorted(by: reverseAlphabetical)
```

- The `sorted(by:)` function has one parameter – a function that accepts two strings and returns a boolean.
- This function is used to compare each pair of elements.

SORTED(BY:) METHOD

```
let cities = ["Albuquerque", "Chicago", "Boston"]

func reverseAlphabetical(_ s1: String, _ s2: String) -> Bool {
    return s1 > s2 // returns true if s1 comes after s2 in the alphabet
}

let reversed = cities.sorted(by: reverseAlphabetical)
```

- Notice that there are two steps: 1) declaring a function to compare two elements, and 2) calling `sorted(by:)`.
- Let's see how we can simplify this...

BASIC CLOSURE

```
// Basic closure syntax.
let reversedCities2 = cities.sorted(by: { (s1: String, s2: String) -> Bool in
    return s1 > s2
})
```

- We can inline the `reverseAlphabetical` function.
- This is called an inline closure (or simply, a closure).

BASIC CLOSURE

```
// Basic closure syntax.
let reversedCities2 = cities.sorted(by: { (s1: String, s2: String) -> Bool in
    return s1 > s2
})
```

Parameters

Return Type

Body

- Parameters and return type are declared *inside* the curly braces
- The keyword `in` is used to mark the start of the function body

TRAILING CLOSURE

```
// Trailing closure syntax.
let reversedCities3 = cities.sorted { (s1: String, s2: String) -> Bool in
    return s1 > s2
}
```

- Because the closure is the last parameter in `sorted(by:)`, we can omit the argument name and parentheses.

INFERRED TYPE

```
// Inferring type from context.
let reversedCities4 = cities.sorted { s1, s2 in return s1 > s2 }
```

- We can omit the closure's parameter and return types and rely on type inference.
- The closure's type will always be `(String, String) -> Bool`.

IMPLICIT RETURN

```
// Implicit return from a single-expression closure.
let reversedCities5 = cities.sorted { s1, s2 in s1 > s2 }
```

- The `return` keyword can be omitted from single-expression (one line) closures.
- As of Swift 5.1, this rule applies to any single-expression function.

SHORTHAND ARGUMENT NAMES

```
// Shorthand argument names.
let reversedCities6 = cities.sorted { $0 > $1 }
```

- If we don’t want to give explicit argument names to our parameters, we can use `$0`, `$1`, and so on as shorthand.

OPERATOR METHOD

```
// Operator method.
let reversedCities7 = cities.sorted(by: >)
```

- The > operator is itself a function.
- It takes two parameters of type String and returns a Bool, so we can use it just like `reverseAlphabetical`.

CLOSURES

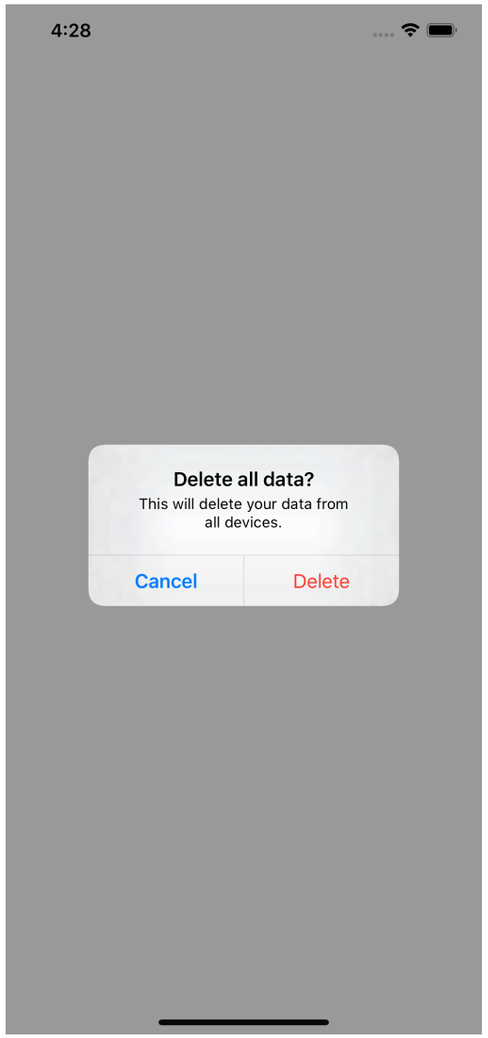
- We've seen lots of ways to write closures in Swift.
- How does this help you create an interactive user experience?

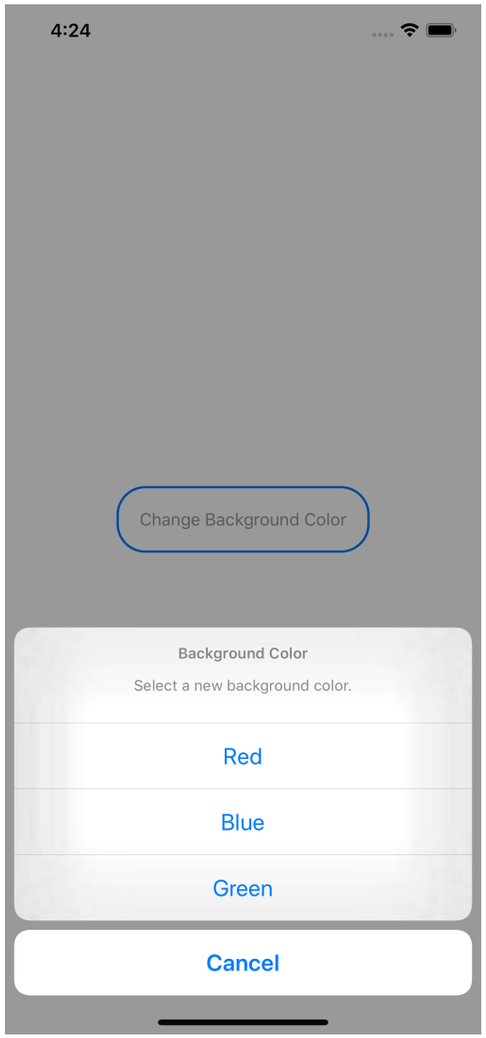
- Example: `UIAlertAction`

ALERT CONTROLLER

- `UIAlertController` is used to display an alert to the user.

ALERT STYLE





ACTION SHEET STYLE

ALERT CONTROLLER

```
// Initialize the alert controller.
let alertController = UIAlertController(title: "Background Color",
                                        message: "Select a color.",
                                        preferredStyle: .actionSheet)
```

- We can initialize the alert controller with a title, message and preferred style (either `.alert` or `.actionSheet`).
- In order to add buttons, we need to create objects of type `UIAlertAction`.

ALERT ACTION

```
// Alert action initializer.
init(title: String?,
     style: UIAlertAction.Style,
     handler: ((UIAlertAction) -> Void)? = nil)
```

- The `handler` parameter's type is a function that takes a `UIAlertAction` and returns void.
- The `?` indicates that this an optional function.
- `= nil` is the default value.

ALERT ACTION

```
// Initialize the "Red" action.
let redAction = UIAlertAction(title: "Red", style: .default) { _ in
    self.view.backgroundColor = .red
}
```

- Notice that we're using trailing closure syntax and inferred types.
- We can use an underscore for the closure's parameter to indicate that we won't be using it.

ALERT ACTION

```
// Add each action to the alert controller.
alertController.addAction(redAction)
alertController.addAction(blueAction)
alertController.addAction(greenAction)
alertController.addAction(cancelAction)

// Show the alert controller on screen.
present(alertController, animated: true)
```

- Once we've created the alert controller and alert actions, we need to add the actions to the controller.
- Last, we present the controller on screen.

CLOSURES

- Summary:
  - Closures are functions. They can be written in-line, rather than being declared ahead of time.
  - `UIAlertAction` is an object that represents a button on an alert.
  - `UIAlertAction`'s initializer takes a closure, which will be executed when the user taps the button.

CLOSURES

- What is printed first, A or B?

```
let redAction = UIAlertAction(title: "Red", style: .default) { _ in
    print("A")
    self.view.backgroundColor = .systemRed
}

print("B")

alertController.addAction(redAction)
present(alertController, animated: true)
```

CLOSURES

- You must be explicit when using `self` inside of closures
- This is to warn you about possible retain cycles
- More on this when we talk about memory management

```
let redAction = UIAlertAction(title: "Red", style: .default) { _ in
    // This should be self.view.backgroundColor...
    view.backgroundColor = .systemRed // !! ERROR !!
}
```

CLOSURES

- We can pass a function to a `UIAlertAction` because functions are first-class citizens in Swift
- This means that functions can be:
  - Passed as arguments to functions
  - Returned from functions
  - Assigned to a variable

CLOSURES

- However, functions are **not** first-class citizens in Objective C.
- For this reason, many controls in UIKit relies on an older pattern called target-action.

PATTERN #2

TARGET-ACTION

TARGET-ACTION

- Imagine you are an Objective C developer...
- Problem: How do you indicate which function should be called when a button is tapped?

TARGET-ACTION

- Imagine you are an Objective C developer...
- Problem: How do you indicate which function should be called when a button is tapped?
- Solution: Provide a unique identifier for the function
  - This is called a selector

TARGET-ACTION

- UIControl has a method called `addTarget`, which allows developers to specify should be called

```
func addTarget(_ target: Any?,
        action: Selector,
            for controlEvents: UIControl.Event)
```

TARGET-ACTION

- The `target` is the class that owns the function that should be called

```
func addTarget(_ target: Any?,
        action: Selector,
            for controlEvents: UIControl.Event)
```

TARGET-ACTION

- The `action` is the selector for the function that should be called

```
func addTarget(_ target: Any?,
        action: Selector,
            for controlEvents: UIControl.Event)
```

TARGET-ACTION

- The `controlEvents` are the events that should triggered the function to be called

```
func addTarget(_ target: Any?,
        action: Selector,
            for controlEvents: UIControl.Event)
```

Tap, swipe, drag, etc.

TARGET-ACTION

```
// Add the target and action for the "touch up inside" event.
button.addTarget(self,
                 action: #selector(changeBackgroundColor(_:)),
                 for: .touchUpInside)
```

- Here is an example of calling `addTarget` on a button

TARGET-ACTION

```
// Add the target and action for the "touch up inside" event.
button.addTarget(self,                    ← Current view controller
                 action: #selector(changeBackgroundColor(_:)),
                 for: .touchUpInside)
```

- The target is the object that implements the function to be triggered.
- In this case, `self` refers to the current view controller.

TARGET-ACTION

```
// Add the target and action for the "touch up inside" event.
button.addTarget(self,
                 action: #selector(changeBackgroundColor(_:)),
                 for: .touchUpInside)
```

Selector

- The action is a selector, which identifies the function to be called when the event occurs.
- In this case, the function is called `changeBackgroundColor(_:)`.

TARGET-ACTION

```
// Function annotated with @objc
@objc func changeBackgroundColor(_ sender: UIButton) {

}
```

- The `@objc` annotation makes the function available to Objective C code
- This is required when using selectors

TARGET-ACTION

```
// Add the target and action for the "touch up inside" event.
button.addTarget(self,
                 action: #selector(changeBackgroundColor(_:)),
                 for: .touchUpInside)
```

UIControl.Event

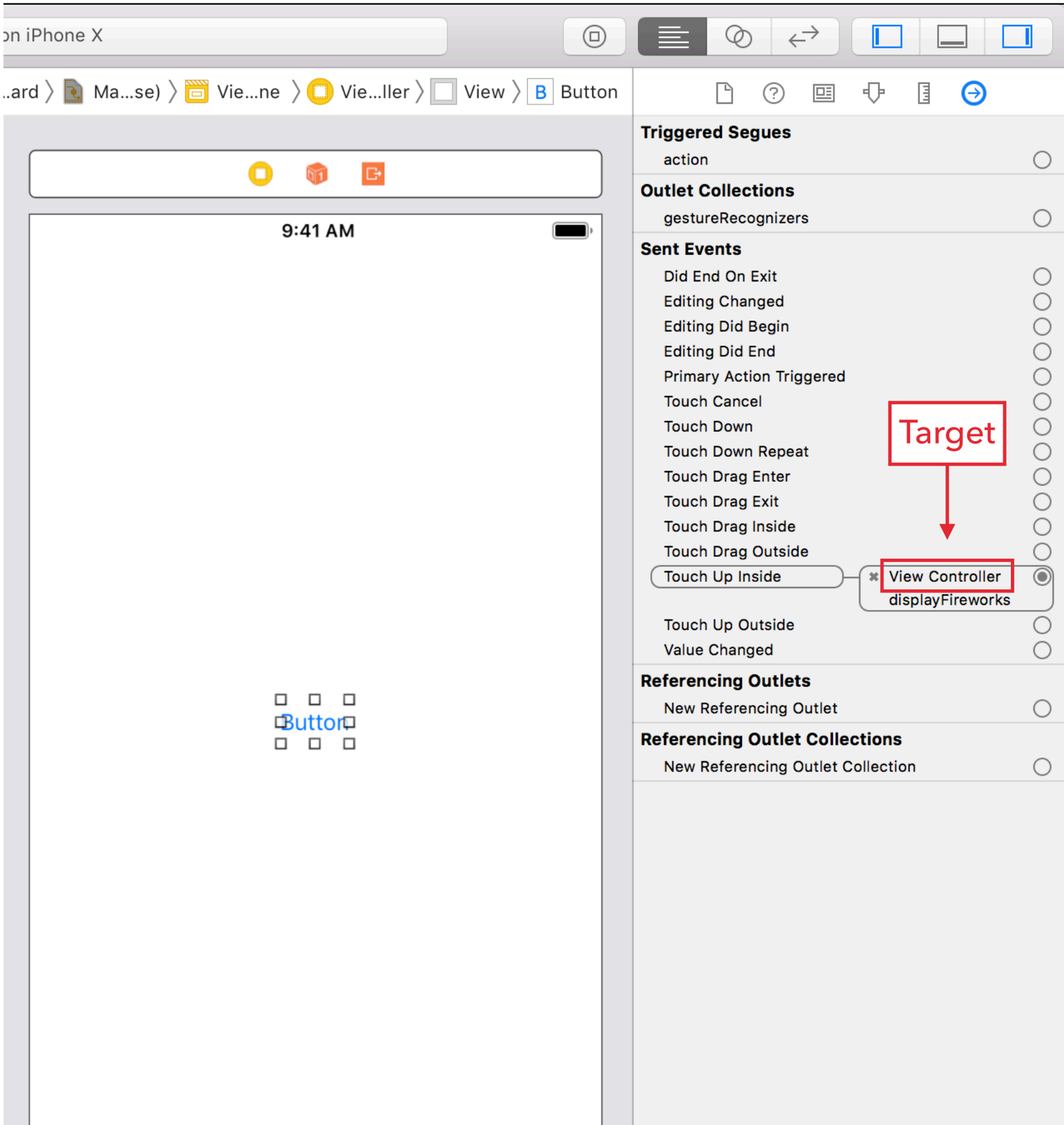
- Here, we're specifying the function that should be triggered when a "touch up inside" event occurs.

CONTROL EVENTS

- There are a lot of different ways for users to interact with controls:
  - `touchDown`
  - `touchUpInside`
  - `touchUpOutside`
  - `valueChanged`
  - `editingDidBegin`
  - `editingDidEnd`
  - And more...

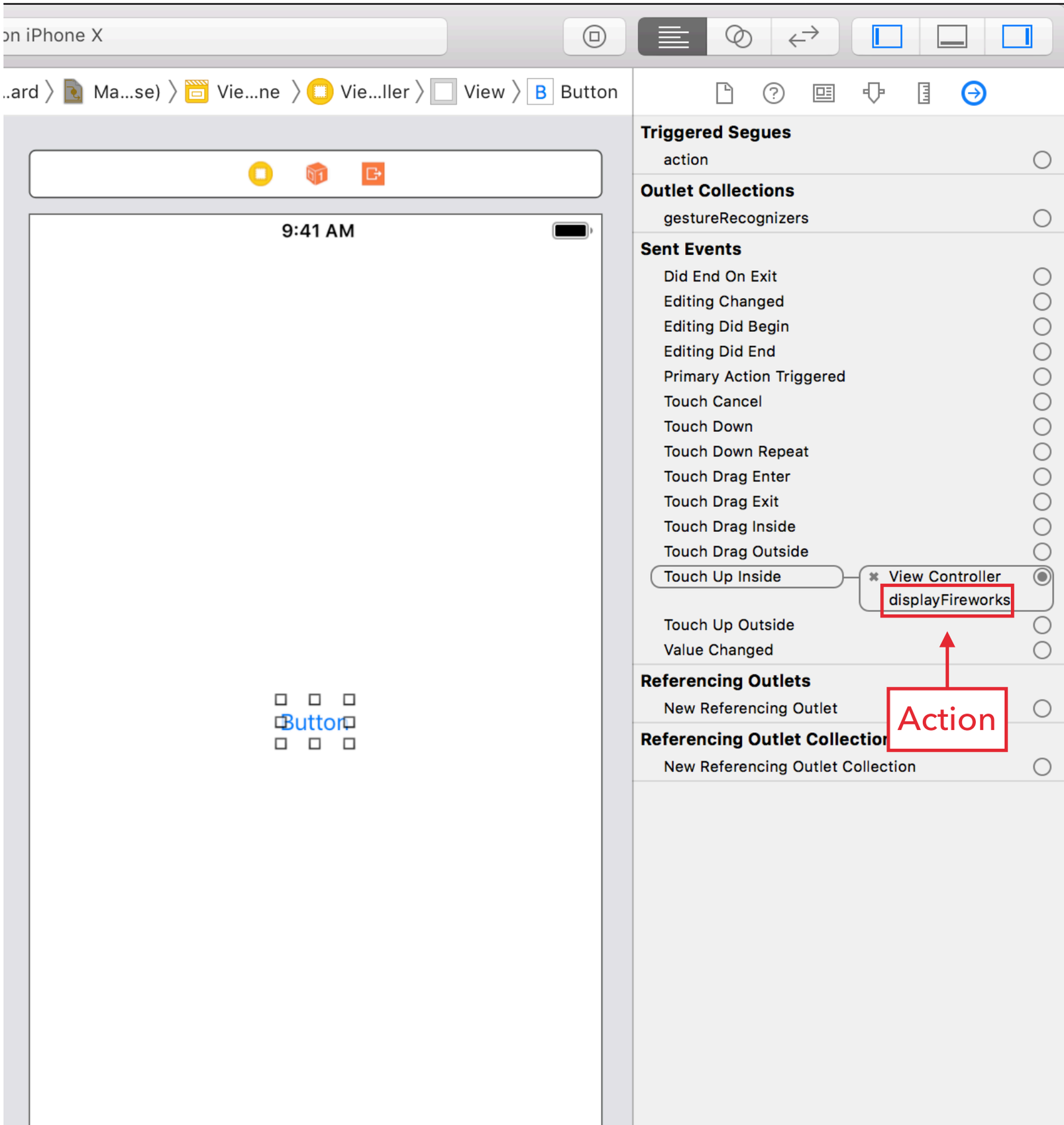
TARGET-ACTION

- This is also the mechanism behind `IBAction`, which is used to connect storyboard elements to code



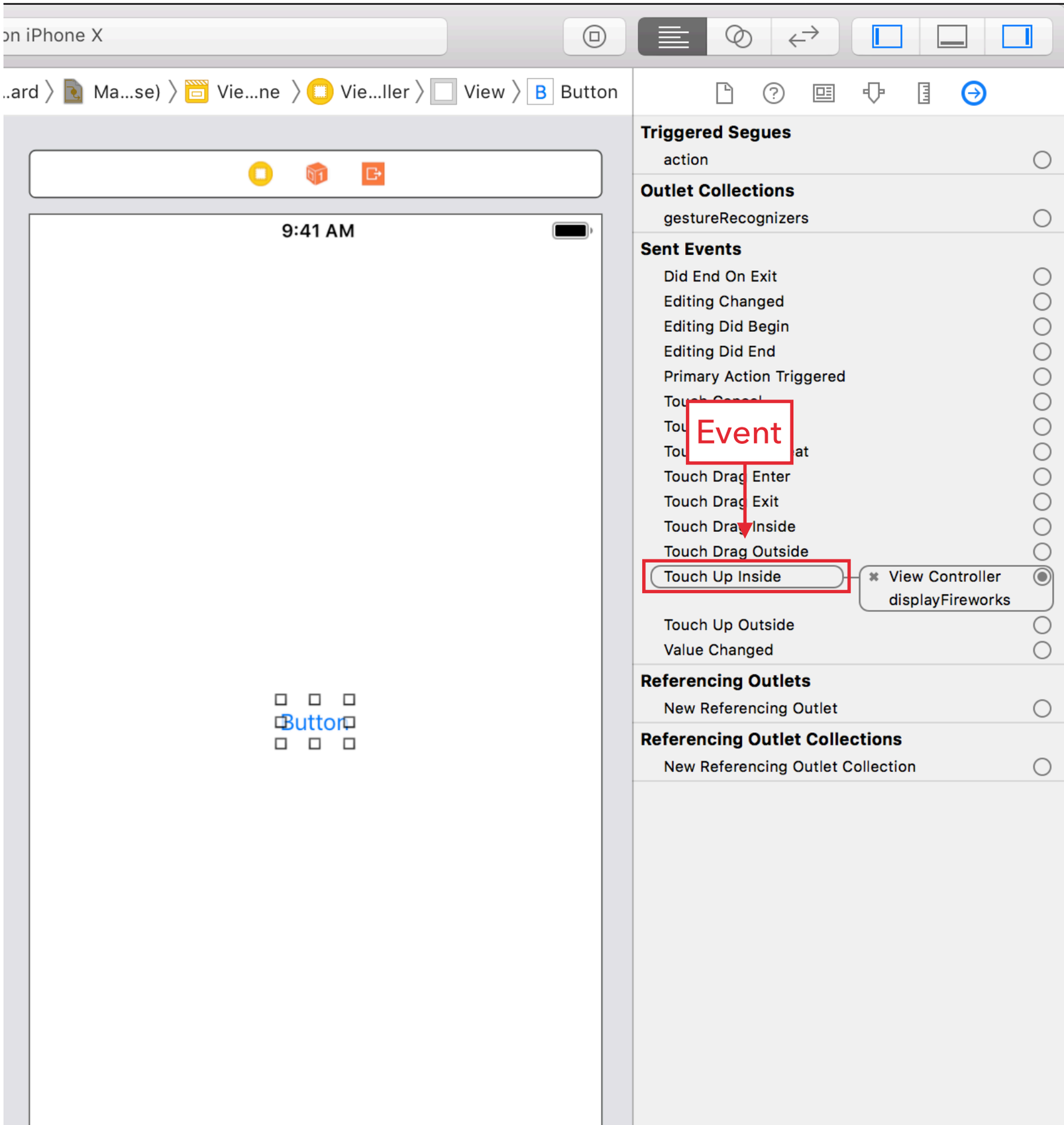
TARGET-ACTION

- This is also the mechanism behind `IBAction`, which is used to connect storyboard elements to code



TARGET-ACTION

- This is also the mechanism behind `IBAction`, which is used to connect storyboard elements to code



TARGET-ACTION

- What happens if you create an `IBAction` and then delete the method from your code without removing the connection from the storyboard?

DOWNSIDES OF TARGET-ACTION

- “Unrecognized selector sent to instance” error
- Limited number of parameters

Listing 1 Action method definitions

```
@IBAction func doSomething()
@IBAction func doSomething(sender: UIButton)
@IBAction func doSomething(sender: UIButton, forEvent event: UIEvent)
```

https://developer.apple.com/documentation/uikit/uicontrol#1658488

PATTERN #3

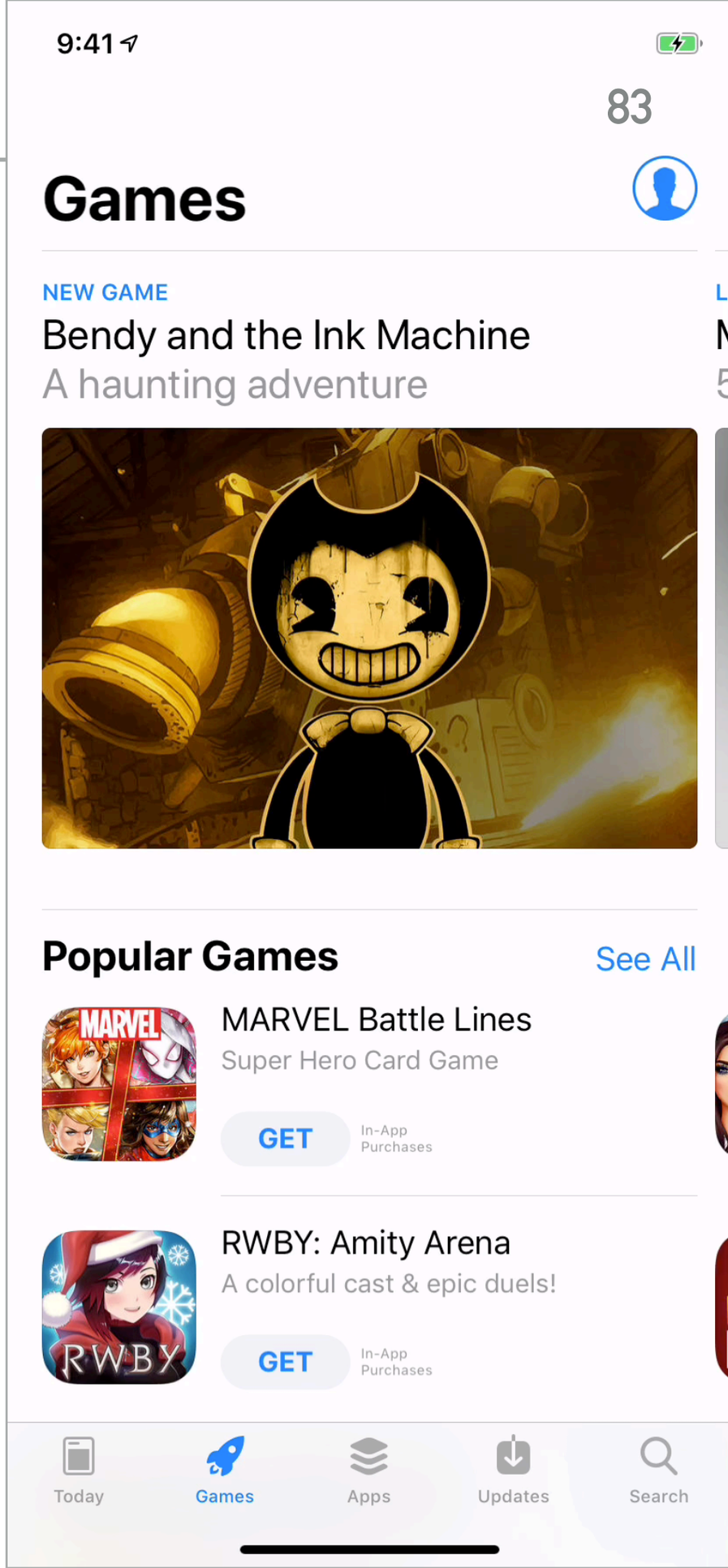
DELEGATES

INTRODUCING SCROLL VIEWS

- “A view that allows scrolling and zooming of its contained views.”

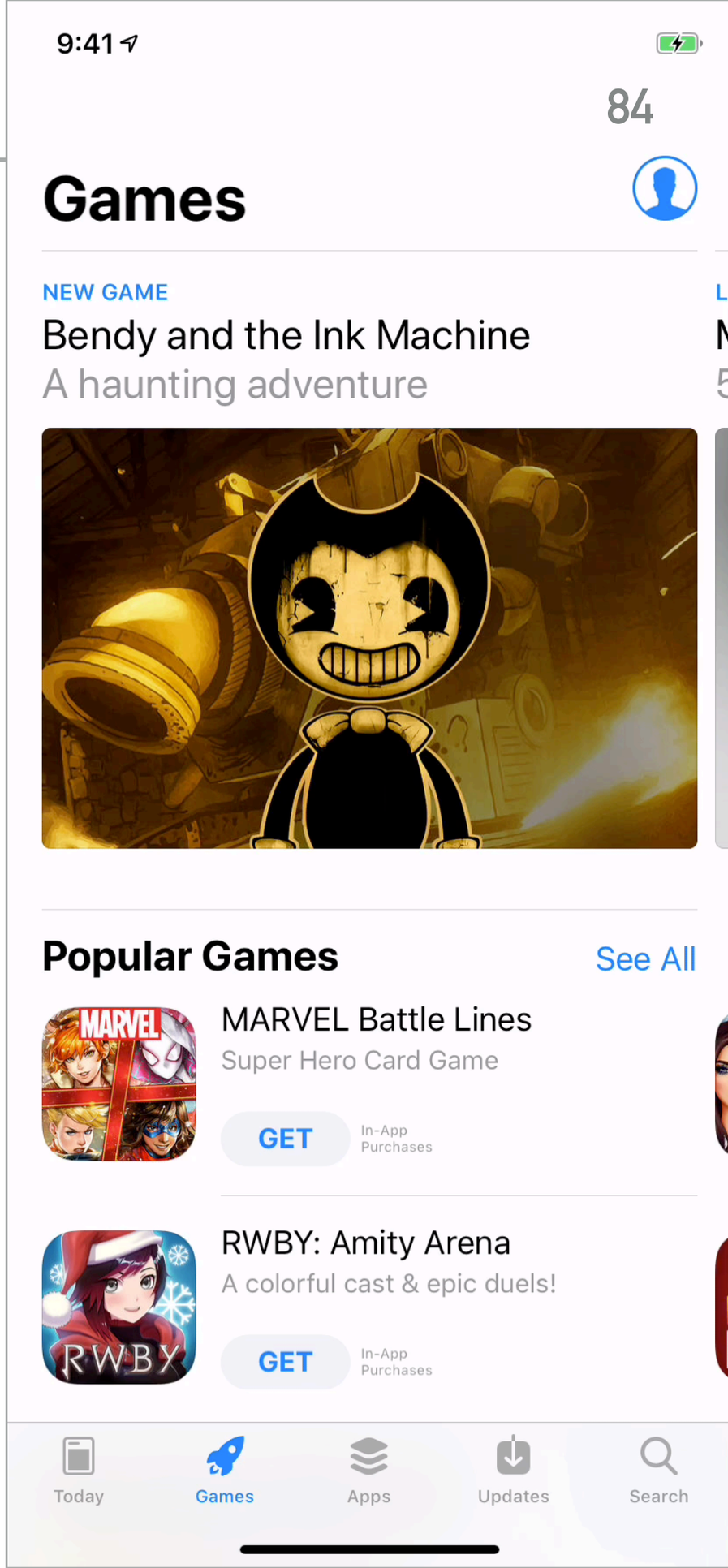
  – UIKit Documentation

- Useful when you have too much content to display at once

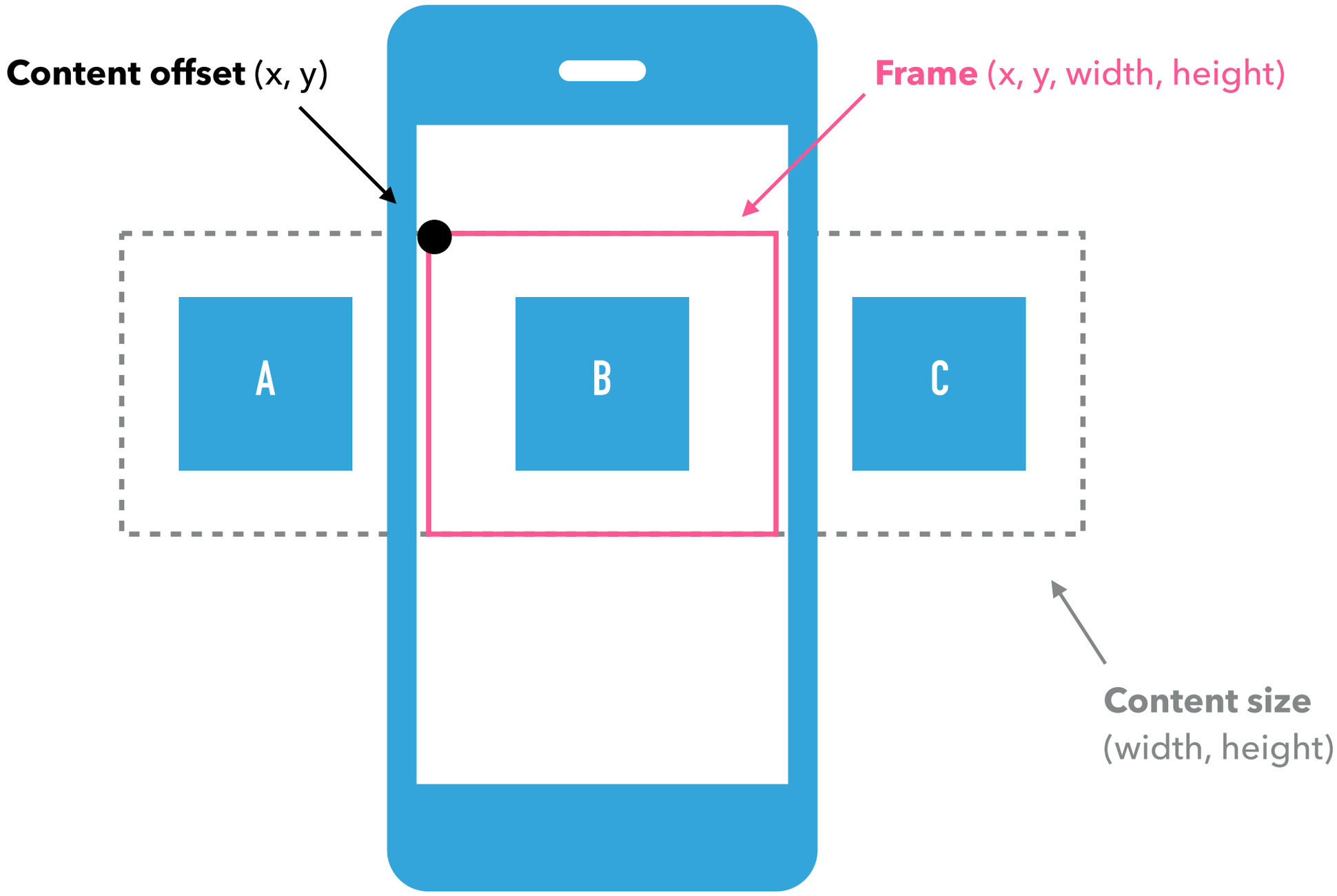


INTRODUCING SCROLL VIEWS

- Scroll views can allow:
  - Vertical scrolling
  - Horizontal scrolling
  - Pinching and zooming

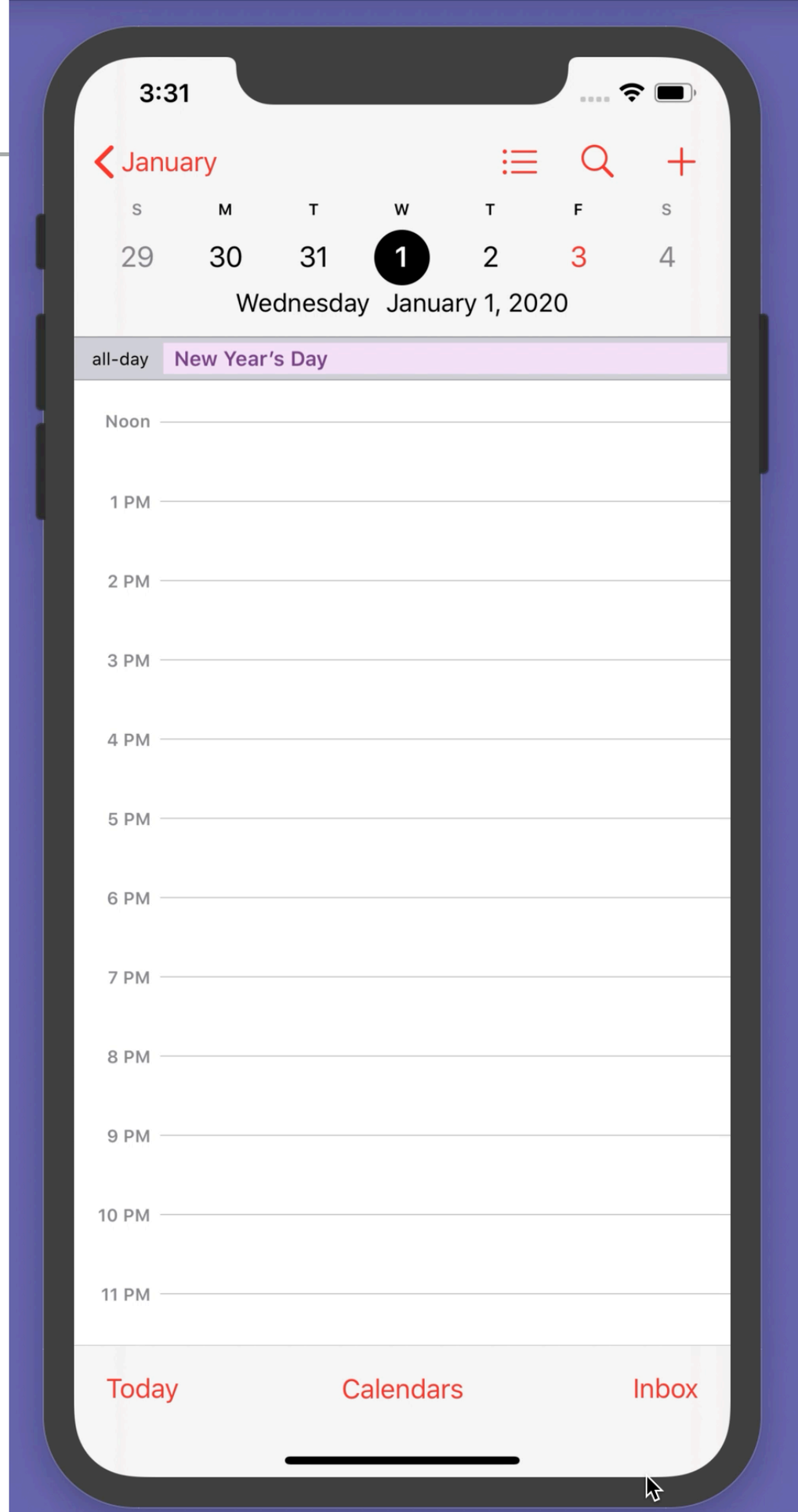


SCROLL VIEW


Content offset (x, y)
Frame (x, y, width, height)
A
B
C
Content size
(width, height)

SCROLL VIEW

- Notice that the date at the top of the screen changes as the user scrolls



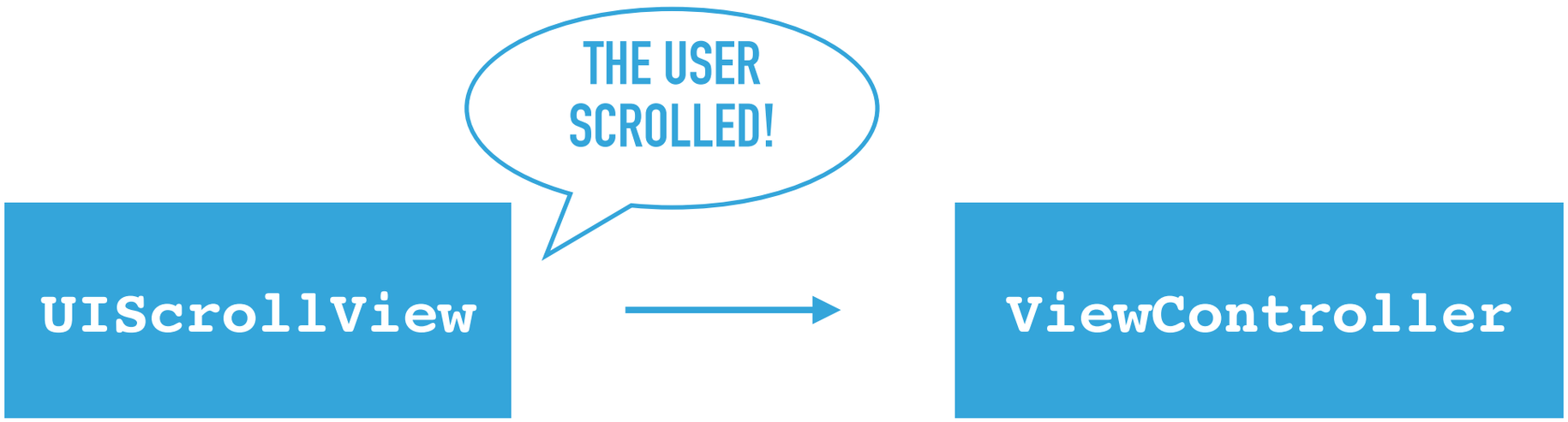
Calendar App

SCROLL VIEW

- An app might need to respond in different ways based on:
    - Scrolling
    - Zooming
    - Decelerating
    - Dragging

SCROLL VIEW

- `UIScrollView` doesn't know what should happen in a particular app when various user interactions occur
- Instead, it delegates responsibility



SCROLL VIEW

- `UIScrollView` has a property called `delegate`
- This property's type is optional `UIScrollViewDelegate`

UIScrollView

```
weak var delegate: UIScrollViewDelegate?
```

SCROLL VIEW

- `UIScrollViewDelegate` is a protocol or blueprint of methods and properties
- Classes, structs and enums can conform to protocols

UIScrollViewDelegate

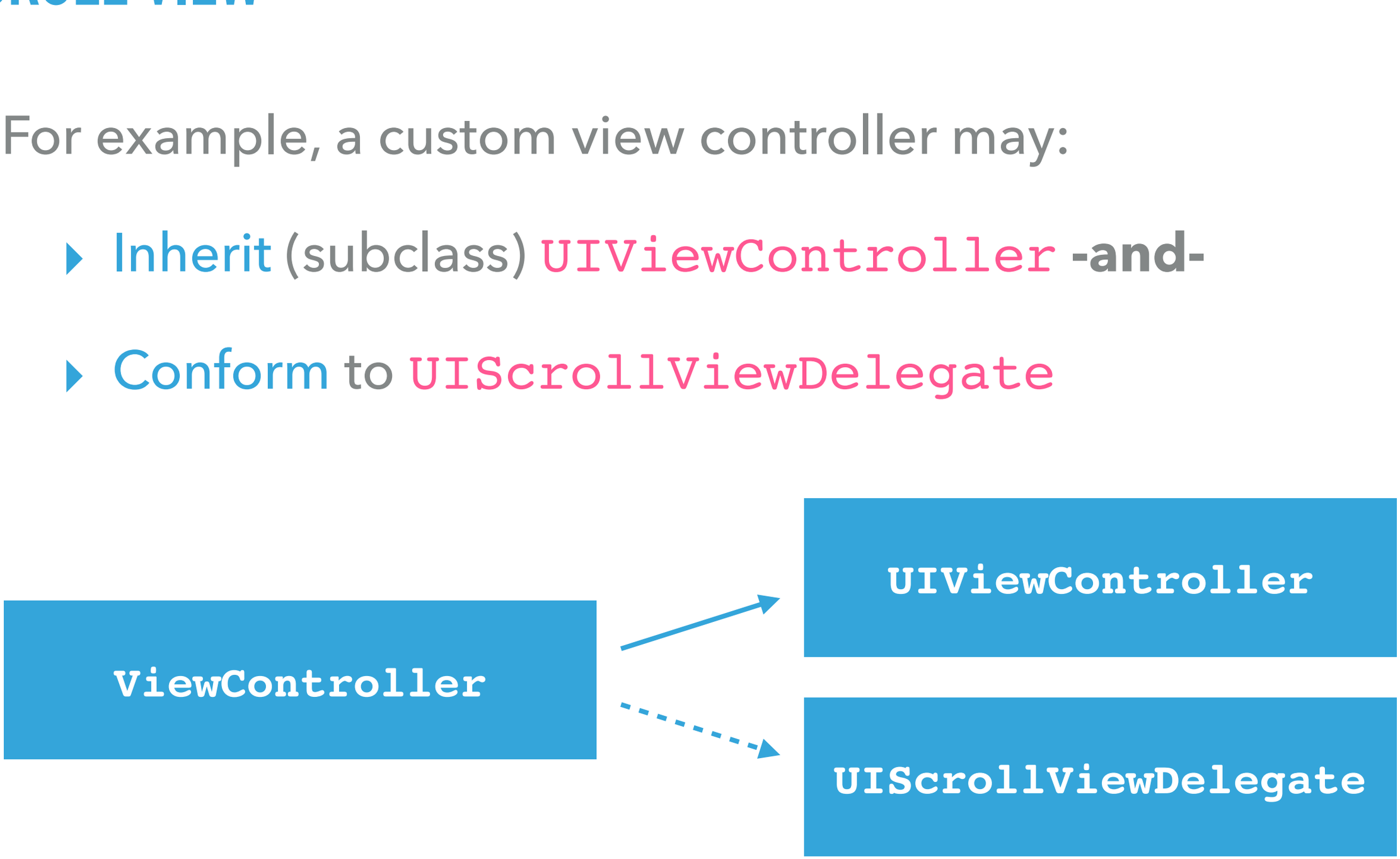
```
func scrollViewDidScroll(_ scrollView: UIScrollView)

func scrollViewWillBeginDragging(_ scrollView: UIScrollView)

...
```

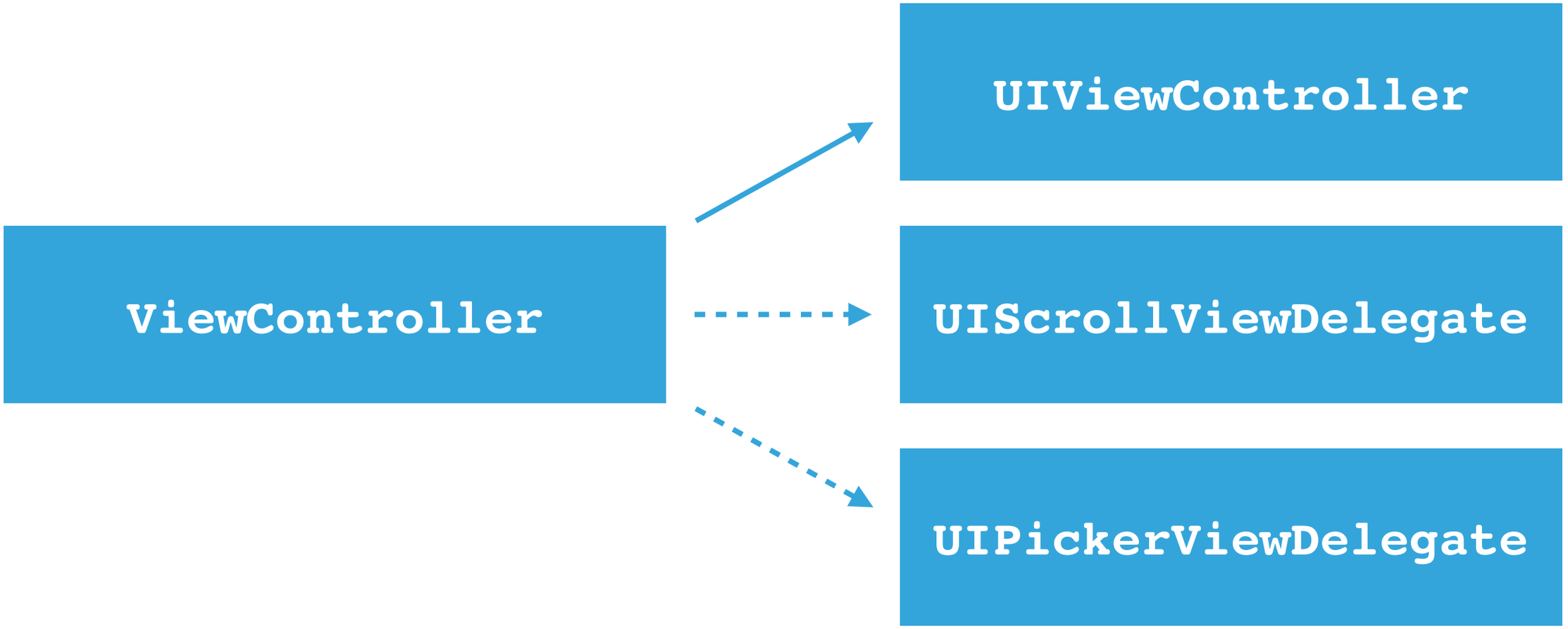
SCROLL VIEW

- For example, a custom view controller may:
  - Inherit (subclass) `UIViewController` **-and-**
  - Conform to `UIScrollViewDelegate`



SCROLL VIEW

- A class may conform to multiple protocols
- However, it can only have one superclass



SCROLL VIEW

```
class ViewController: UIViewController, UIScrollViewDelegate {

}
```

- List the protocols a class conforms to when defining the class
- Note that the superclass comes before any protocols

SCROLL VIEW

```
extension ViewController: UIScrollViewDelegate {

}
```

- Or use an extension to specify protocol conformance
- This is a nice way to keep your code organized

SCROLL VIEW

```
extension ViewController: UIScrollViewDelegate {

    func scrollViewDidScroll(_ scrollView: UIScrollView) {
        print(scrollView.contentOffset)
    }
}
```

- Implement the protocol methods
- Some methods are required, others are optional

SCROLL VIEW

```
class ViewController: UIViewController {

    @IBOutlet var scrollView: UIScrollView!

    override func viewDidLoad() {
        super.viewDidLoad()
        scrollView.delegate = self
    }
}
```

- Set the view controller (`self`) as the scroll view's delegate

SCROLL VIEW

```
// Somewhere inside the code for UIScrollView
delegate?.scrollViewDidScroll?(self)
```

- When an event occurs, the scroll view will call the appropriate method on its delegate
- Note that the delegate and the method are both optional

IB VS. CODE

BUILDING INTERFACES IN INTERFACE BUILDER

- Pros:
  - Simple & fast
  - Visual
- Cons:
  - Merge conflicts hard to resolve 🧐
  - Not always flexible enough

BUILDING INTERFACES IN CODE

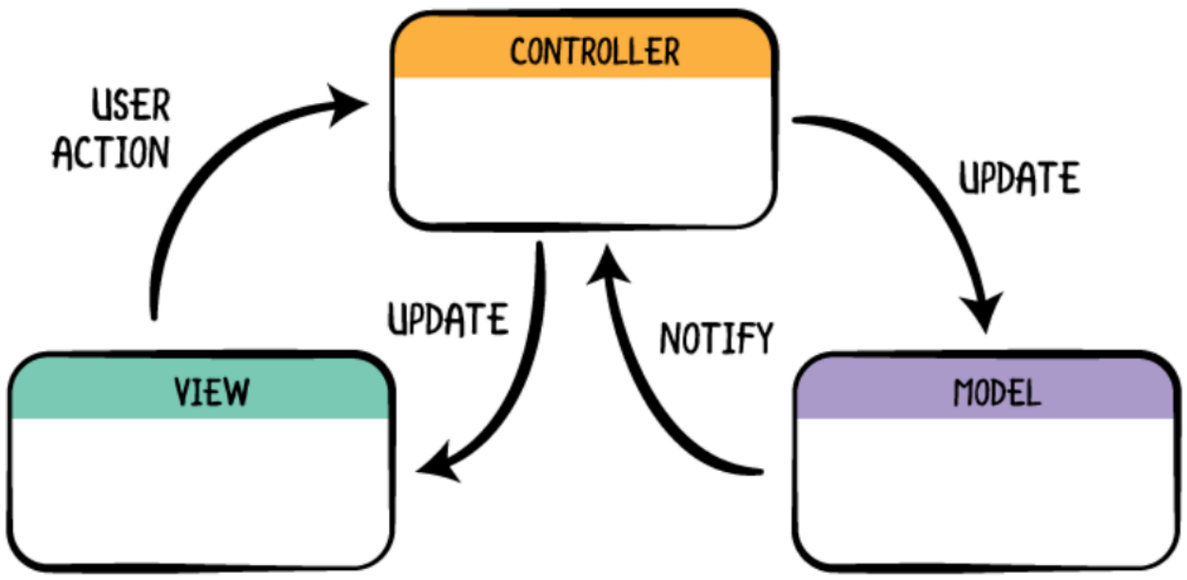
- Pros:
  - More flexibility & control
  - Merge conflicts easier to resolve
- Cons:
  - No immediate visual feedback

#1 iOS DESIGN PATTERN

MVC

MVC

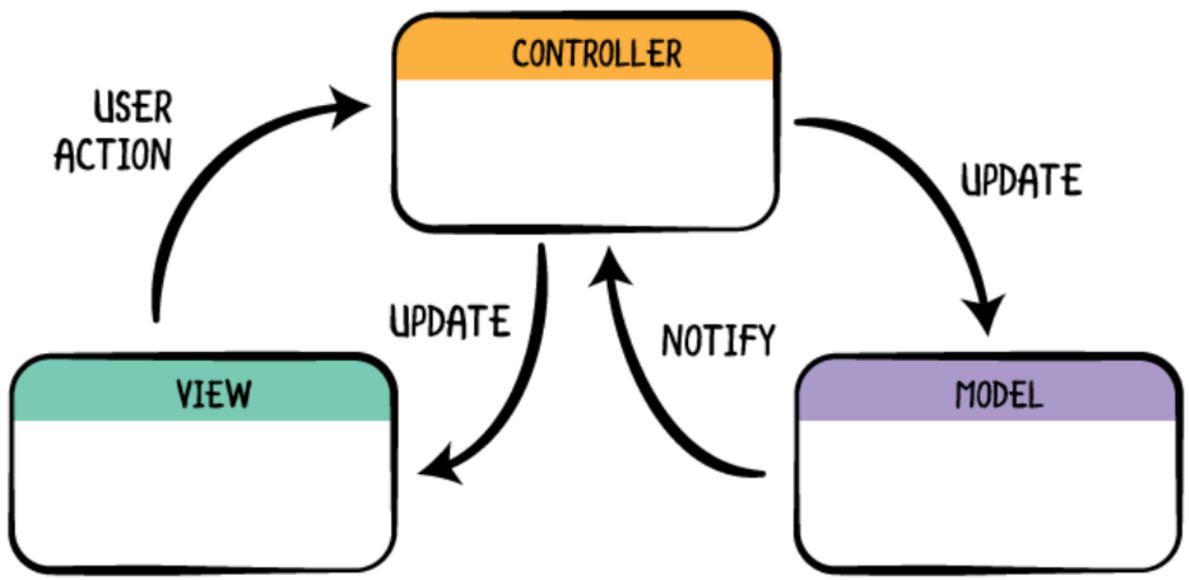
- MVC = Model-View-Controller
- The central design pattern of iOS development



https://www.raywenderlich.com/1073-model-view-controller-mvc-in-ios-a-modern-approach

MVC

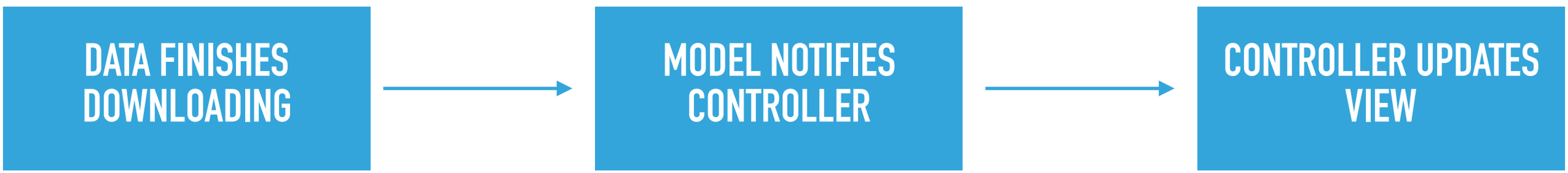
- Model: data objects
- View: visual components
- Controller: coordinates communication between model and view



https://www.raywenderlich.com/1073-model-view-controller-mvc-in-ios-a-modern-approach

EXAMPLES


USER TAPS "LIKE" BUTTON
VIEW NOTIFIES CONTROLLER
CONTROLLER UPDATES MODEL


DATA FINISHES DOWNLOADING
MODEL NOTIFIES CONTROLLER
CONTROLLER UPDATES VIEW

MASSIVE-VIEW-CONTROLLER?

- There's a tendency for view controllers to become HUGE
- We'll talk about ways to address this throughout the course

iOS APPLICATION

LIFECYCLE

IN THE EARLY DAYS

- Before iOS 4, an application could be in one of two states:
  - Running
  - Not running
- Tapping the app icon on the home screen would launch the app
- Tapping the home button would quit the app

iOS 4 AND LATER

- Starting in iOS 4, the effect of tapping the home button changed
- Instead of terminating the app, the app is simply suspended
- Tapping the app icon of a suspend app resumes the app (rather than launching it from scratch)

iOS 4 AND LATER

- Of course, an app *can* be terminated if one of the following happens:
  - The user powers off the device
  - The user force-quits the app from the app switcher
  - The system decides to kill the app

APPLICATION STATES

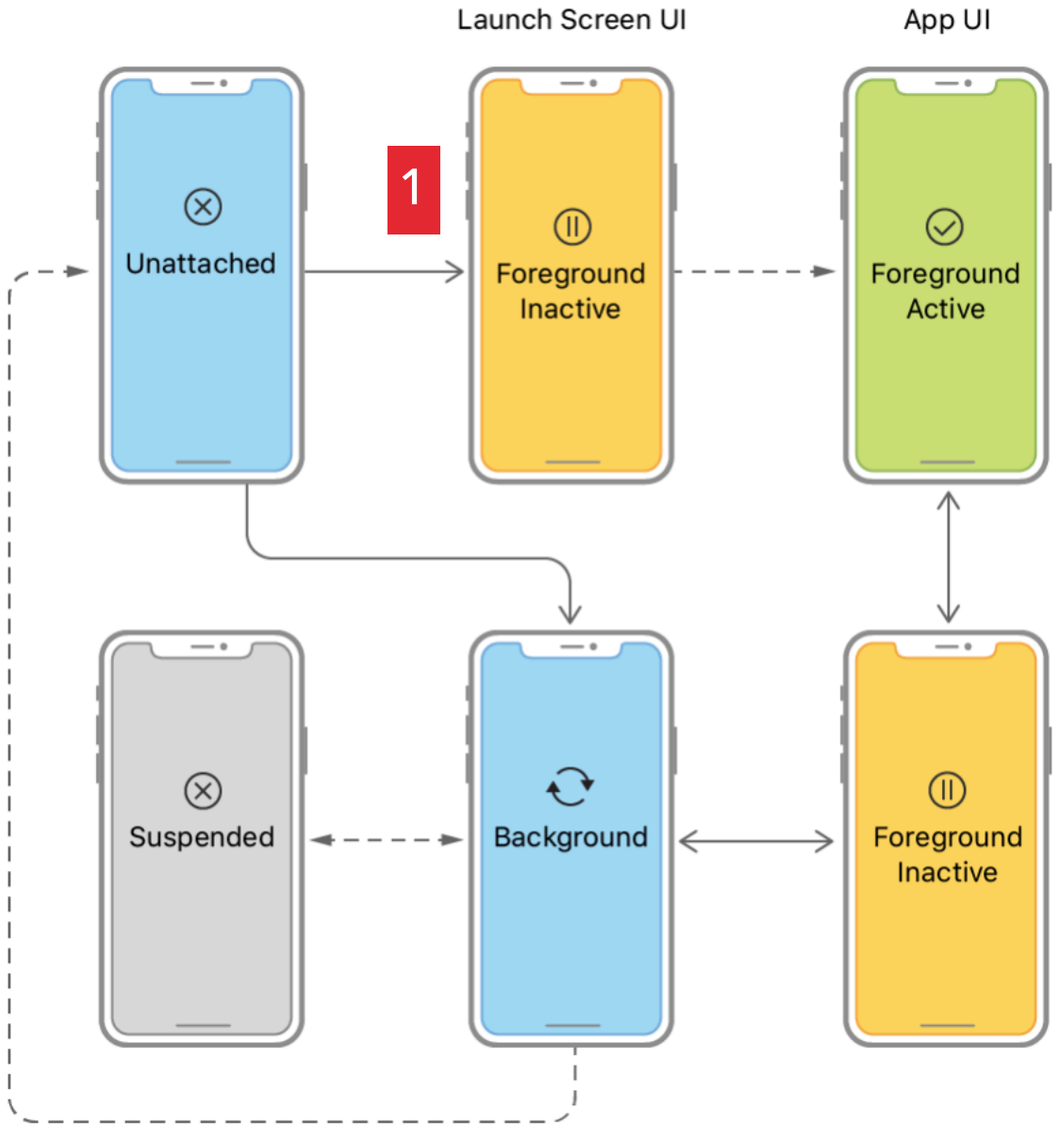
- Newer versions of iOS have complicated things even more:
    - Apps can be backgrounded, but *not* suspended
    - Suspended apps can be woken up briefly, remaining in the background
    - Apps can be inactive without being backgrounded

SCENE-BASED APPS

- On iOS 13, iPad apps can support multiple scenes
  - A scene is one instance of your app's UI
  - One scene might be in the foreground, while others are backgrounded or suspended
- Xcode's templates create scene-based apps by default

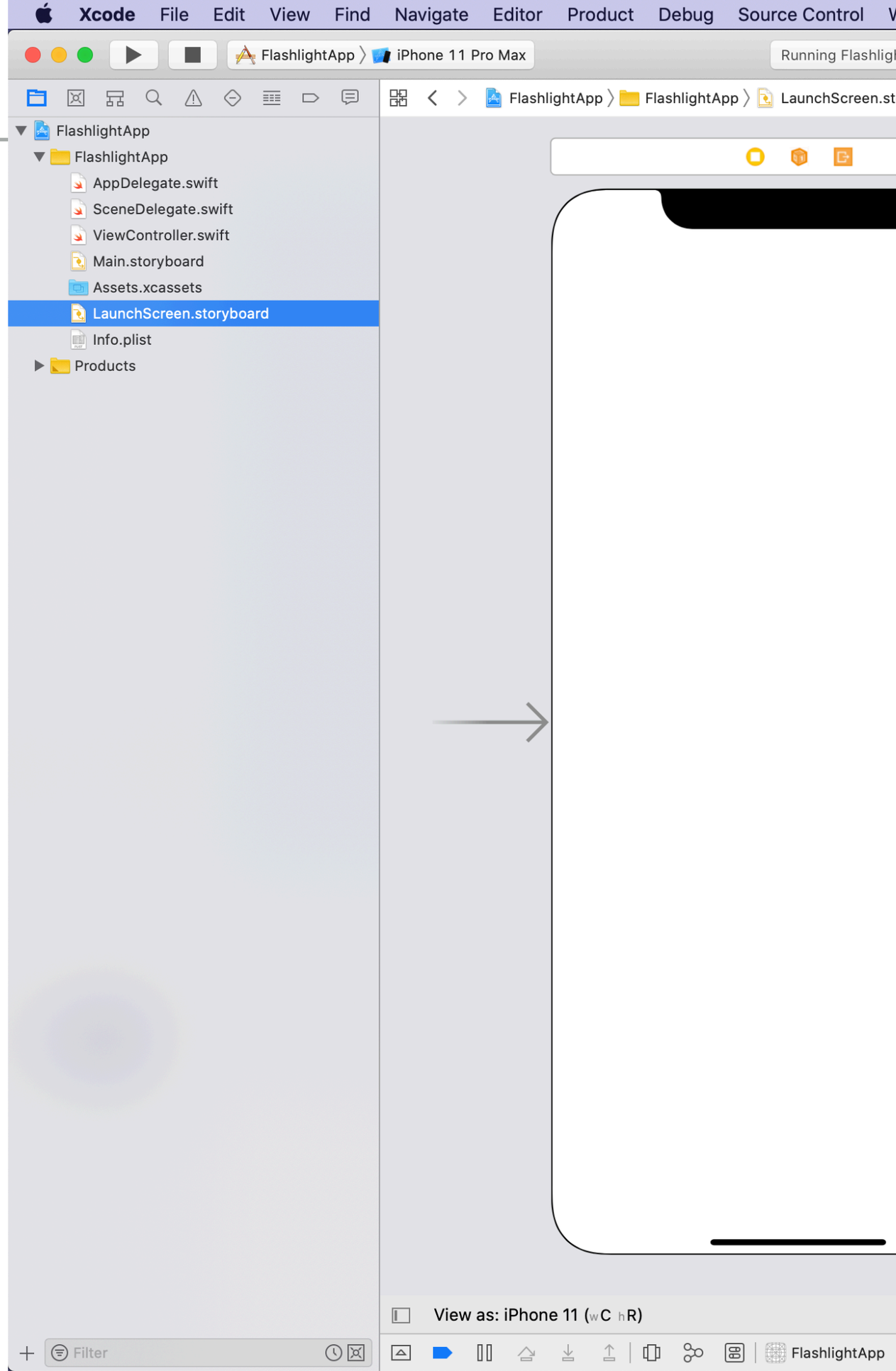
LIFECYCLE

- First, the scene briefly enters the foreground inactive state
  - The system will display LaunchScreen.storyboard
- This must contain static content only



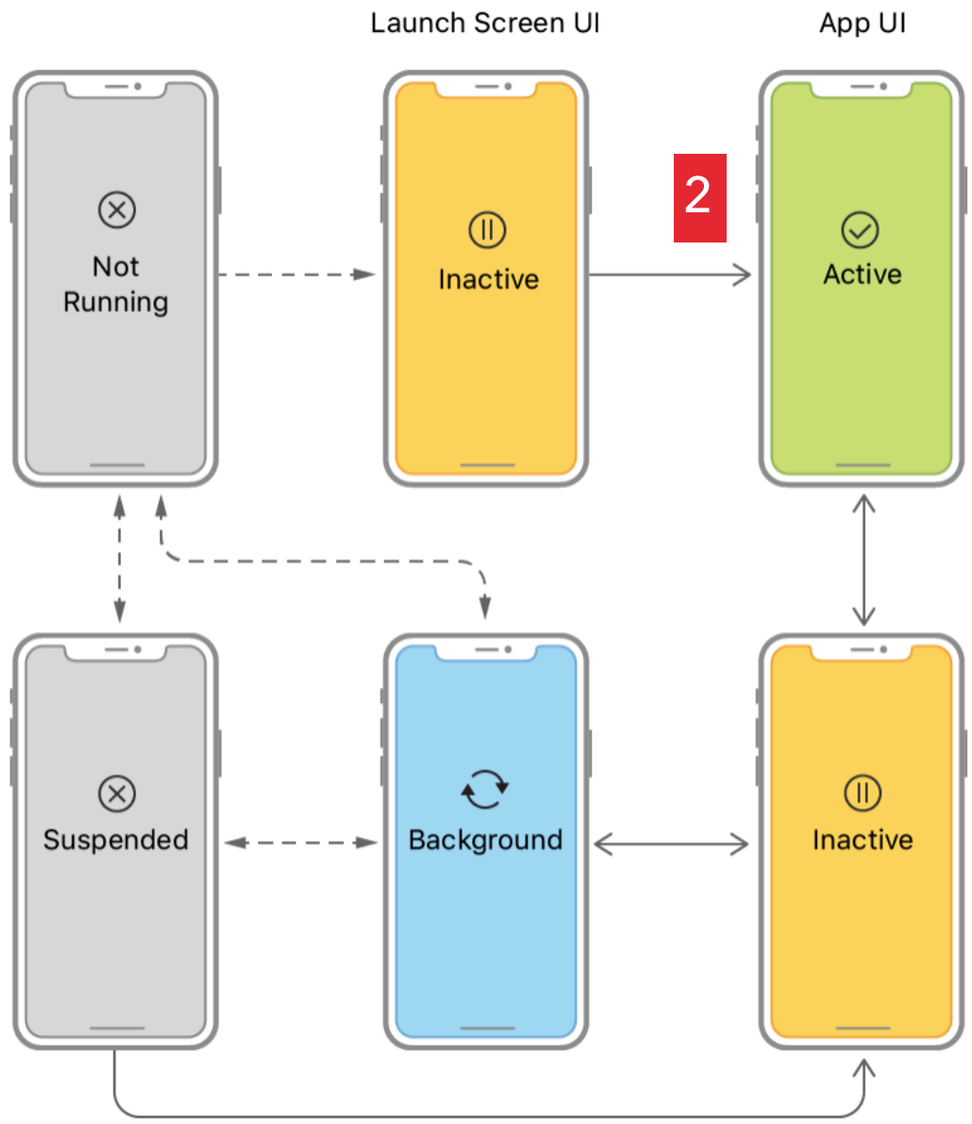
LIFECYCLE

- LaunchScreen.storyboard



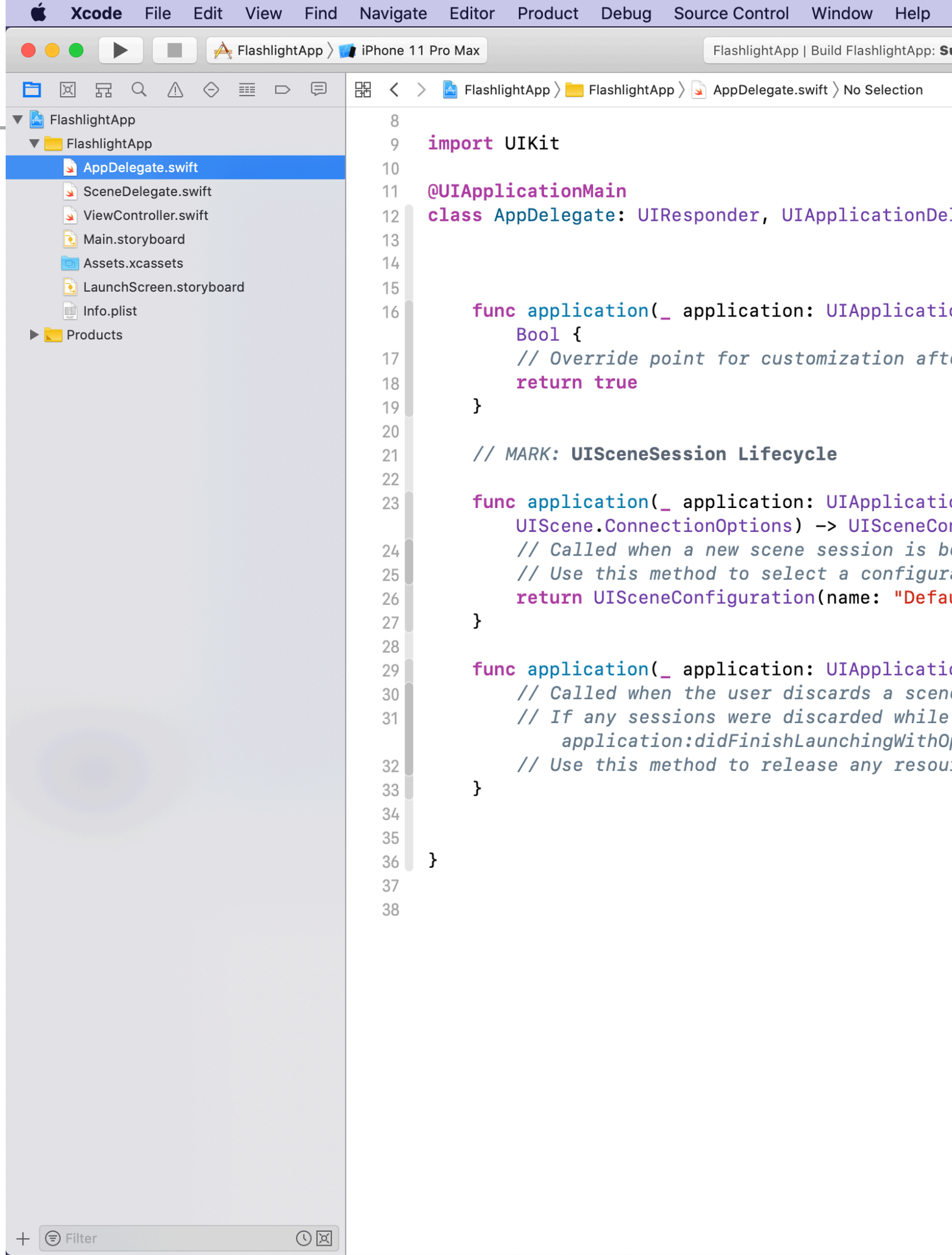
LIFECYCLE

- Next, the scene becomes active
  - `AppDelegate` and `SceneDelegate` methods are called
- Main.storyboard's initial view controller is displayed



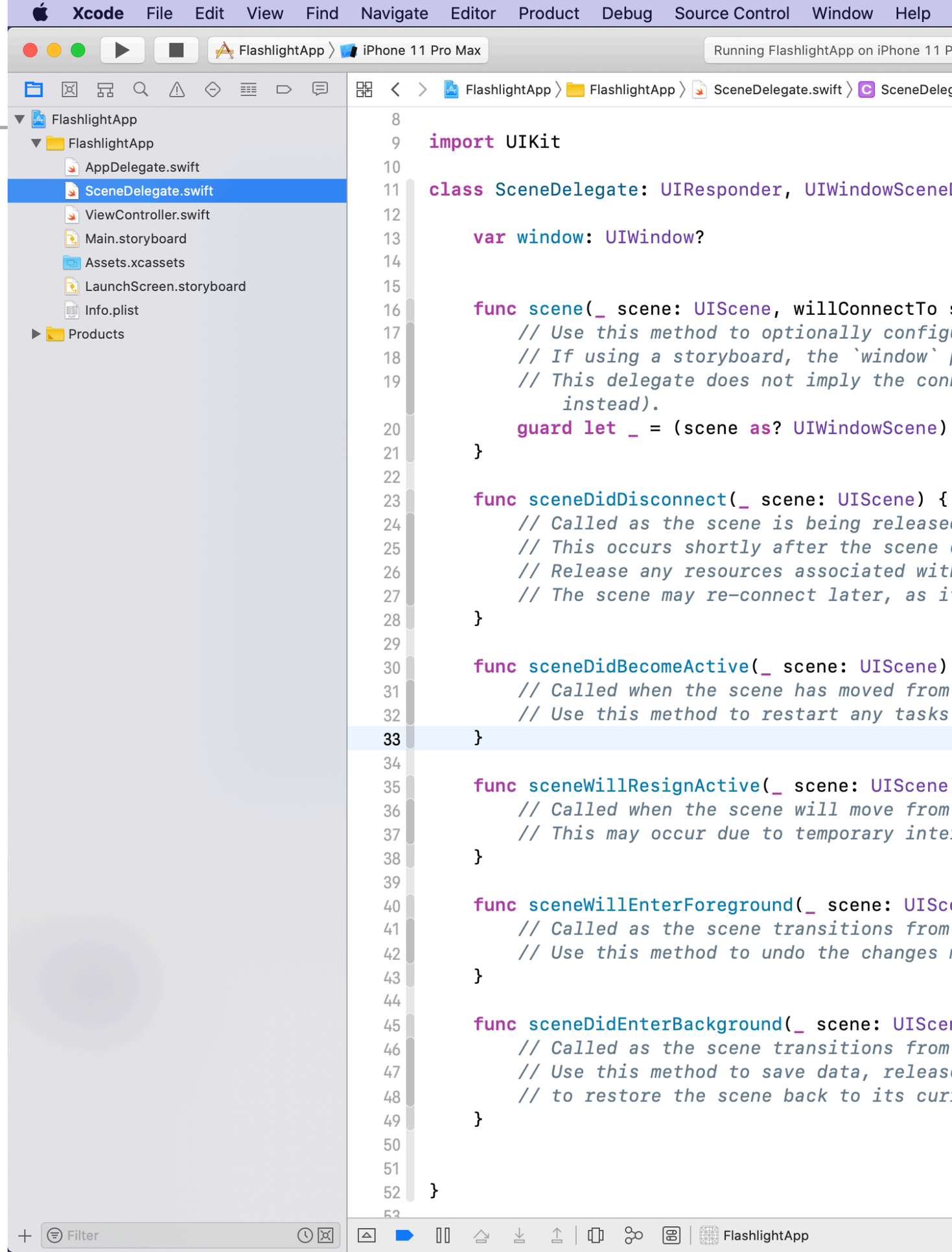
LIFECYCLE

- AppDelegate
- Notified when the app launches and when scenes are created or destroyed



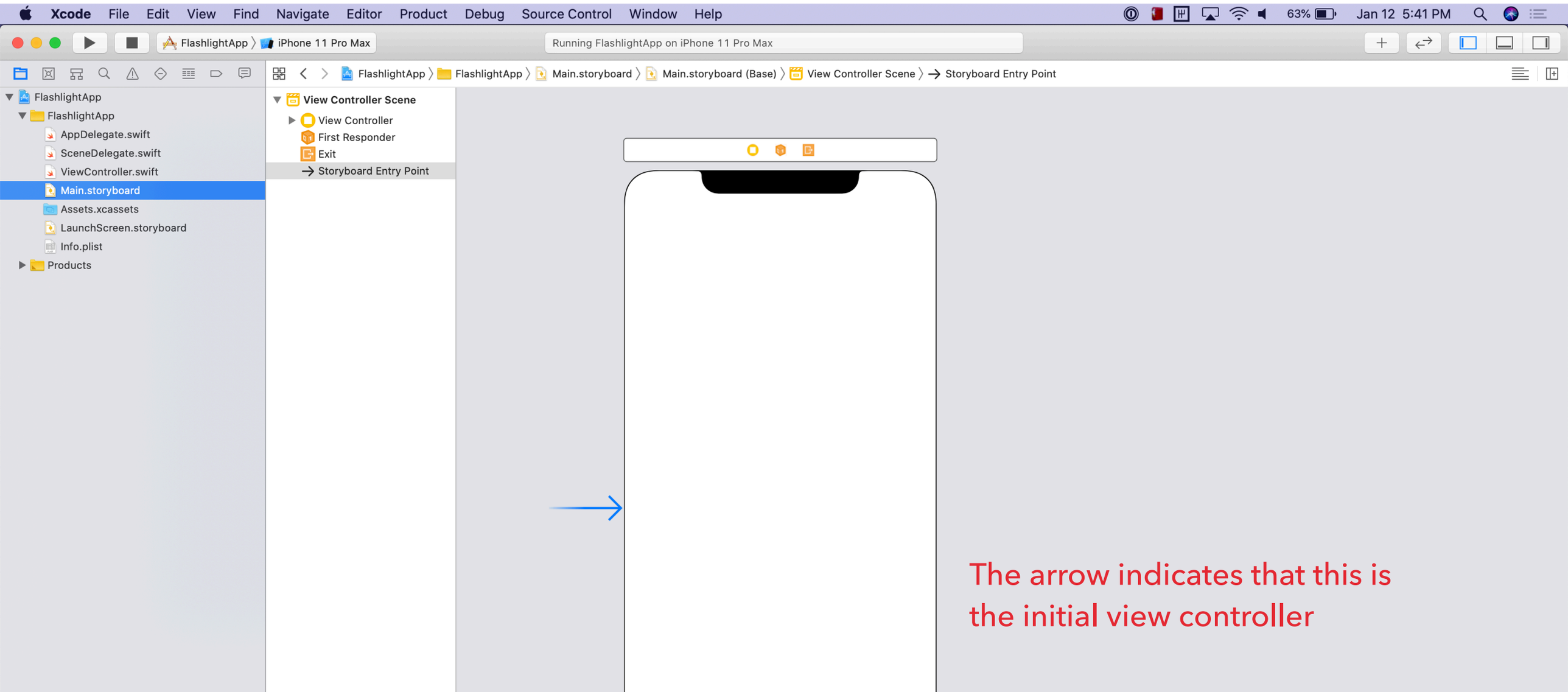
LIFECYCLE

- SceneDelegate
- Notified when scenes move from active to inactive and background to foreground



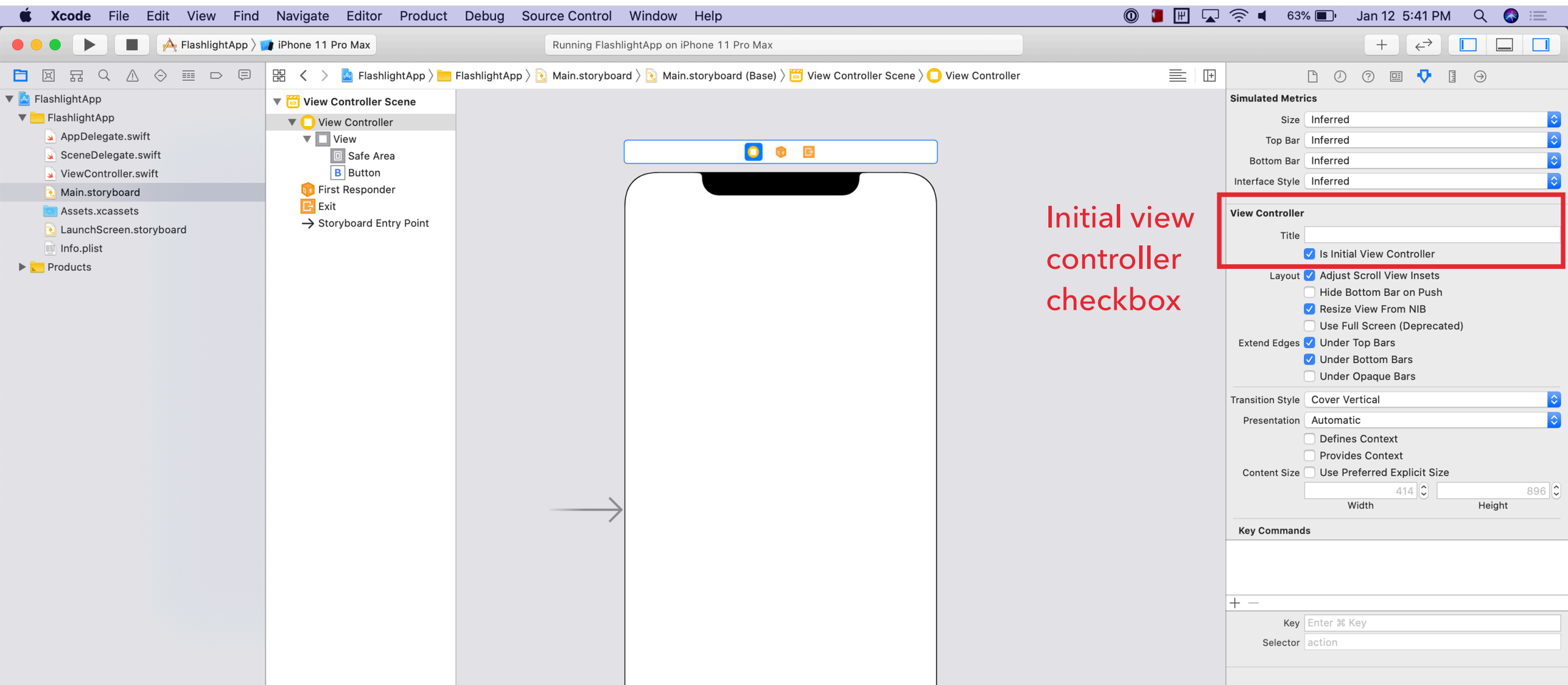
LIFECYCLE

- Main.storyboard's initial view controller is displayed first



LIFECYCLE

- Main.storyboard's initial view controller is displayed first



ASSIGNMENT #2

IT'S A ZOO IN THERE

IT'S A ZOO IN THERE

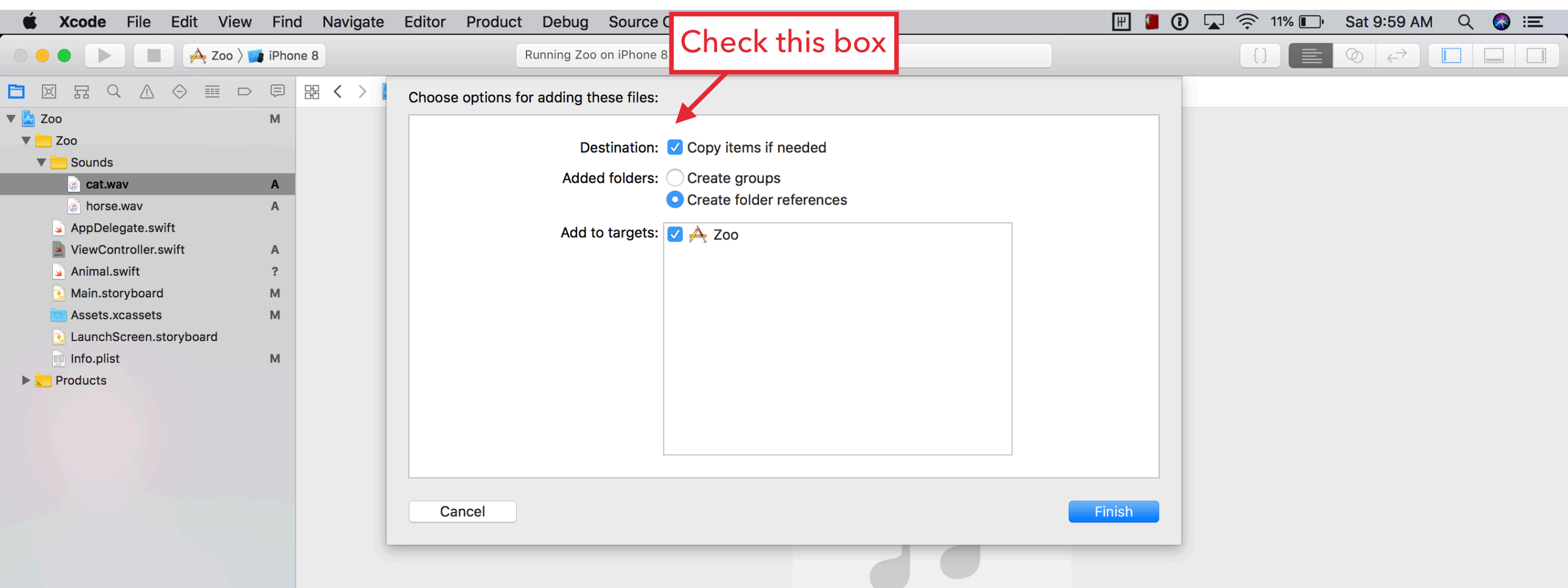
- Build an app that displays images, sounds and information about 3 animals of your choice
- We will use:
  - `UIButton` (target-action)
  - `UIAlertController` (closures)
  - `UIScrollView` (delegates)
  - Interface Builder + programmatic layout

IT'S A ZOO IN THERE

- There are a few things we didn't cover in class, like playing sounds, that you'll need to read about on your own.

IMPORTANT!

- You must check “Copy items if needed” for sounds to be included in your repository



IT'S A ZOO IN THERE

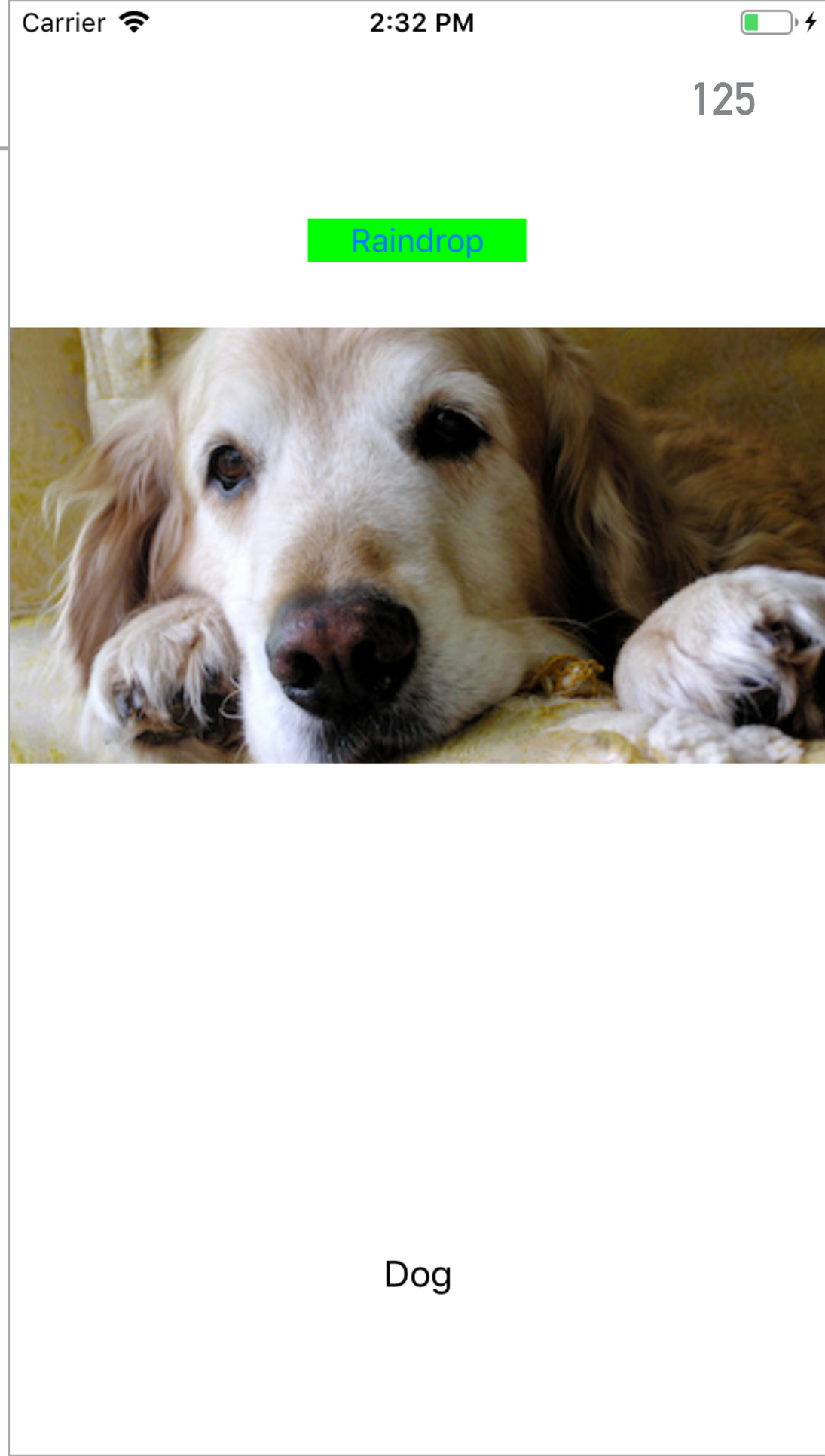
- Due Wednesday, January 22 at 5:29pm
- Post any questions to Slack

THE SECRET ART OF

VIEW DEBUGGING

SET THE BACKGROUND COLOR

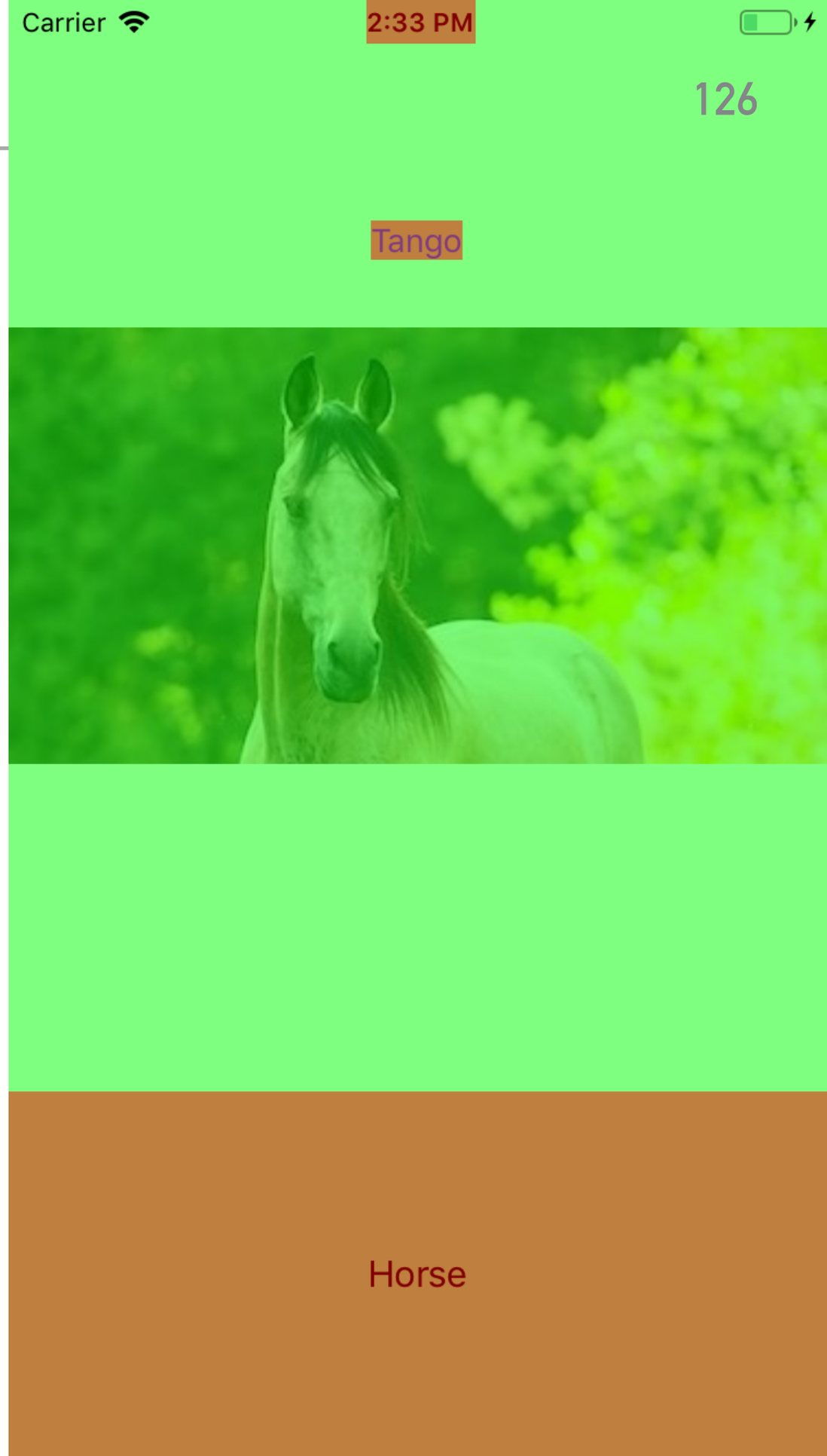
- Easy way to visualize exact frame of your view



COLOR BLENDED LAYERS

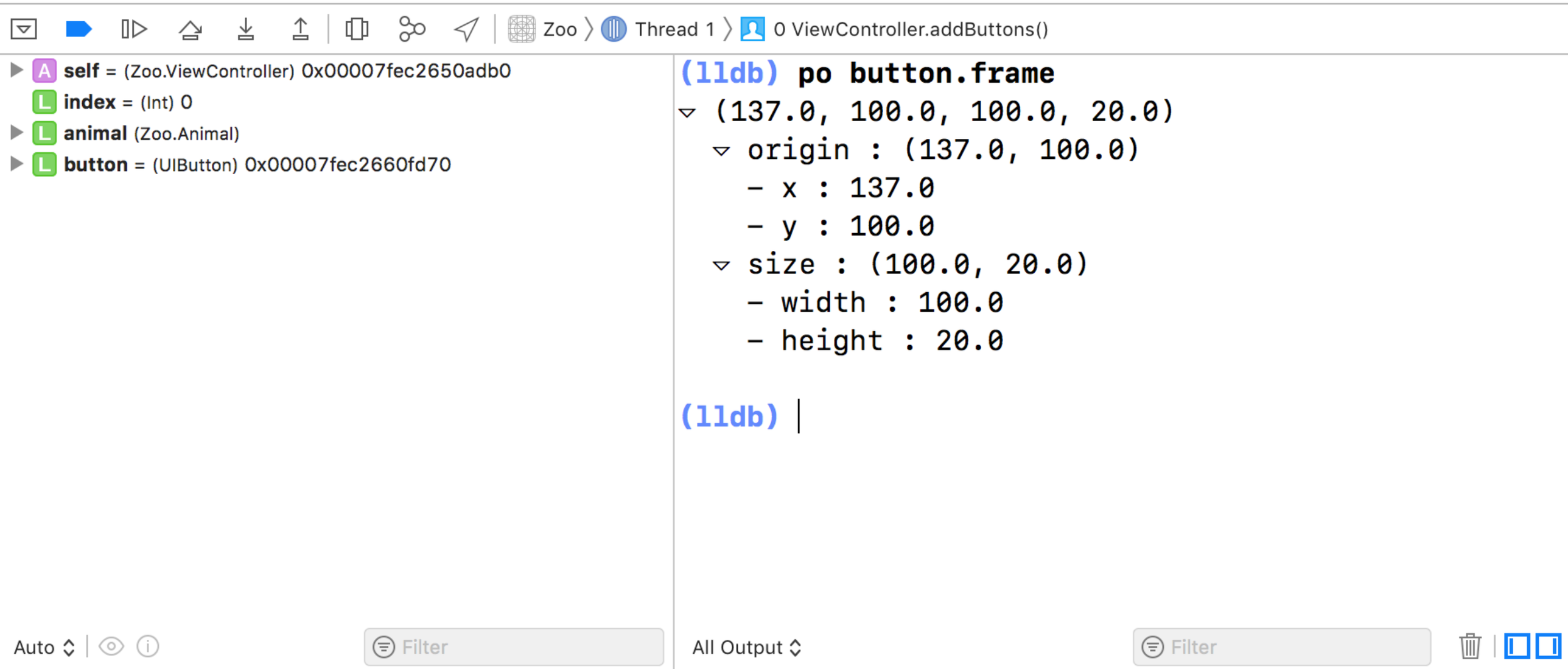
- In the simulator, go to

  Debug –> Color blended layers



PRINT A VIEW'S FRAME

- Start by setting a breakpoint and running your app



USE THE VIEW DEBUGGER